Module 4: Mutation

Topics:

•Mutation of simple variables

•Memory model for Scheme

•Mutation of structures

Readings: HtDP 34, 35, 36, 40.3, 40.5, 41.1

Mutation: What is it?

- Mutation: Changing an identifier's value
- Standard part of many programming languages (incuding Python)
- DrRacket Language Level:

  Advanced Student Scheme

No Mutation in Beginner Scheme

A rule (so far in Scheme):

Any function always produces the same value when called with the same arguments

```
(define x 120)
(sqr x)   => 14400
… ;; any other code
(sqr x)   => 14400
```

Sometimes mutation would be helpful…

- A gps program to map between two locations
  - Might be helpful if it could alter route if new roads are added or roads are blocked
- A program for a board game
  - Information must change as the players take turns
- A program that involves reading input from the user via the keyboard
  - What happens depends on which key was pressed

How are these applications different?

- The need to incorporate time into the computation
- The need to incorporate interaction with the user
- The need to change the state of the application

➔ We need mutation for many applications

Doesn't this make things more complicated?

- Yes!
  - It can be harder to trace a program
  - It can be harder to debug a program
  - We need to know the value of our identifiers throughout execution

Mutation in Scheme

- The keyword **`set!`** allows us to change the value of an identifier

```
(define avg 87)
avg                     => 87
…
(set! avg 88.6)
avg                     => 88.6
```


How to use `set!`

`(set! v expr)`

- **`v`** must be a previously-defined identifier
- **`v`** cannot be a parameter
- can be inside a local expression or function
- **`expr`** can be any Scheme expression
- **`expr`** is evaluated first, and then value of **`v`** is updated

Value of `set!` expression

- Produces nothing, represented by **`(void)`**
- It has an *effect*: the value of an identifier is changed
- The order of evaluation is now critical.

Tracing programs with mutation

```
(define x 34)
```

x $\boxed{34}$

```
(set! x (- 2.8 200))
```

x $\boxed{\begin{array}{c}\cancel{34}\\ -197.2\end{array}}$

Tracing a (slightly) bigger program

```
(define child-age 12)
child-age
(set! child-age 14)
child-age
(set! child-age
      (add1 child-age))
child-age
```


Yet another mutation trace

```
(define num 5)
(define L
    (list num
          (set! num (+ 20 num))
          num))
num
```

Placing a mutation like this is poor programming style. Don't do it!

Functions and mutation

Within a function, **`set!`** can be used to change

- Identifiers declared outside the function (called state or global variables)
- Identifiers declared locally inside the function (called local variables)

But

- Cannot be used to change parameter values

Mutation and the Design Recipe

- Contract:
  – May use **`(void)`** as the produced value
- Purpose:
  – Describe the produced value as before
  – If no produced value, simply use
  `;; Produces (void)`

Mutation and the Design Recipe

- *New section:* Effects
  – If the function includes a mutation, this section must now explain what is changing
  – Use variable and parameter names where suitable to describe the changes
- *Modified section:* Examples
  – Might need to use English (in addition to Scheme) to explain what happens when the function is called

```
;; drop-x: num -> (void)
;; Purpose: produces (void)
;; Effects: reduces the value of x
;;     by reduction
;; Examples: if x has the value 10,
;;     and we call (drop-x 4),
;;     then x will have the value 6
(define (drop-x reduction)
   (set! x (- x reduction)))
(define x 20)
(drop-x 20)
x
(drop-x 100)
x
```


Sequences of code

- The body of a function can be either:
  - A mutation, or
  - An expression that produces a value.
- What if we want to mutate multiple identifiers as well as produce a value?
  - Need a new type of expression: **begin**

begin

(begin ex_1 ex_2 … ex_N)

- Execute **ex_1** first
- Then **ex_2**
- …
- Finally, **ex_N**
- value of the **begin** expression is the value of **ex_N**

What's happening here?

```
(define n 5)
(begin (set! n (+ n 10))
       (sqr n))
```

```
(define m 5)
(define p 2)
(begin (sqr m)
       (set! m (+ m 10))
       (set! p (* m p)))
```

```
(local [(define z 5)]
  (begin
       (set! z (+ z 10))
       (sqr z)))
```


Using `begin` in a function

```
;; how often has convert-lbs-to-kg been called?
(define count-calls 0)

;; convert-lbs-to-kg: num[>=0] -> num[>=0]
;; Purpose: Produces the number of kg in lbs
;; Effects: Increases count-calls by 1
;; Example: If count-calls is 3,
;;   (convert-lbs-to-kg 2.2) => 1 and
;;   sets count-calls to 4
(define (convert-lbs-to-kg lbs)
  (begin
    (set! count-calls (add1 count-calls))
    (/ lbs 2.2)))
```


How to test a mutation?

First attempt: The usual way …

```
;; Testing drop-x
(define x 20)
(drop-x 20)
(check-expect x 0)
(drop-x 100)
(check-expect x -100)
```

This doesn’t work!

A new approach to testing mutation

```
;; initialization:
;; define variables needed for testing
(check-expect
   (begin ;; step 1 – set testing values
          ;; step 2 – call function
          ;; step 3 – determine result
          ;;          of test
   )
   ;; provide expected value of
   ;; begin expression
   expected)
```


Testing drop-x

```
(define (drop-x reduction)
   (set! x (- x reduction)))
;; Initialization
(define x 20)
(check-expect (begin (set! x 20)
                     (drop-x 20)
                     (equal? x 0))
              true)
(check-expect (begin (set! x 0)
                     (drop-x -5)
                    x)
              5)
```


Testing convert-lbs-to-kg

```
;; Step 0: Initialization
(define count-calls 0)
(define answer 0)
;; Testing convert-lbs-to-kg
(check-expect
 (begin
   ;; Step 1: Set state variable
   (set! count-calls 0)
   ;; Step 2: Call function
   (set! answer (convert-lbs-to-kg 2.2))
   ;; Step 3: Compare actual and expected values
   (and (= answer 1)(= count-calls 1)))
 ;; Expect boolean result to be true
 true)
```

Understanding our Scheme Programs

- Previously, to fully understand our program, we applied simplification steps to determine its final value.
- Now, concerned with "state of the program"
  - The collection of all variables and their values
  - It can change!
  - => We need a memory model for more complicated data values (structures, lists, functions)

Our Memory Model

- Defining a variable:
  - Sets up a box in memory, labeled with the identifier
- Initializing or mutating a variable:
  - Atomic values (numbers, strings, characters, boolean, symbols)
    - The value is put into the labeled box
  - More complex data values (structures, lists, functions)
    - The value is stored elsewhere
    - The labeled box contains an reference to where the value is stored (a pointer or arrow from the box to the location in memory)

About "complex" values

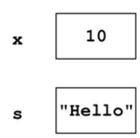
- We maintain a box (called **`make-x`**) which contains all the structures created in our program
- When a new structure is created with a **`make`** call, it is added to the box, with an arrow referencing it.
- List values and function values have separate boxes as well.

What is happening in memory?

```
(define x 10)
(define s "Hello")
(define p1 (make-posn 2 4))
(define-struct posn3d (x y z))
(define p2 (make-posn3d 1 -2 8))
(set! x -4)
(set! s #\a)
(set! p1 'red)
(set! p2 (make-posn 0 0))
```

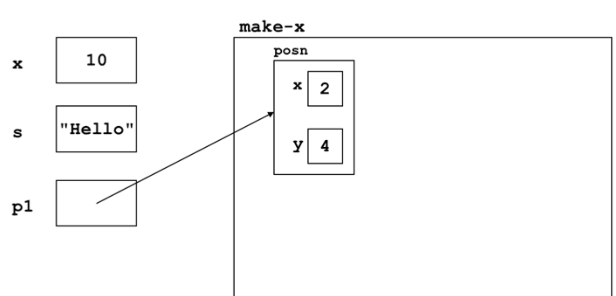

What is happening in memory?

```
(define x 10)
(define s "Hello")
```



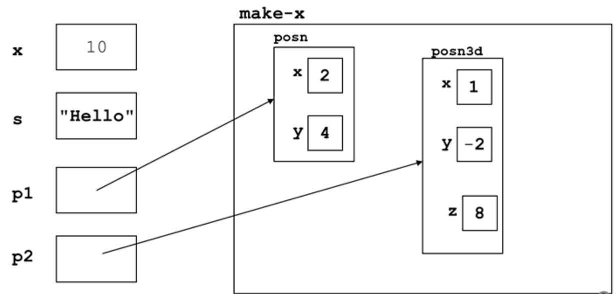
What is happening in memory?

```
(define p1 (make-posn 2 4))
```



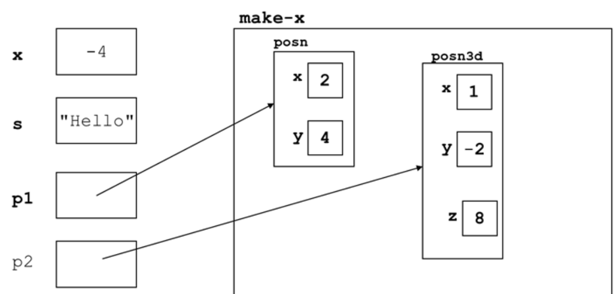
What is happening in memory?

```
(define-struct posn3d (x y z))
(define p2 (make-posn3d 1 -2 8))
```



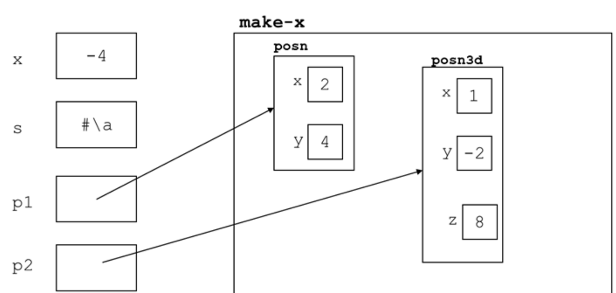
What is happening in memory?

```
(set! x -4)
```



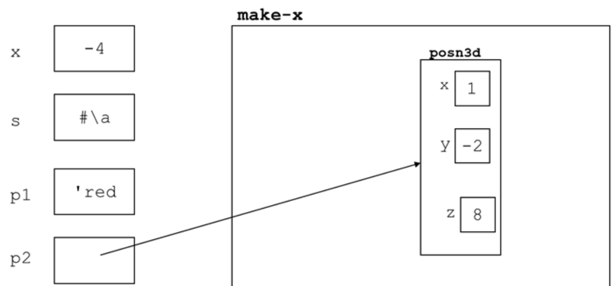
What is happening in memory?

```
(set! s #\a)
```



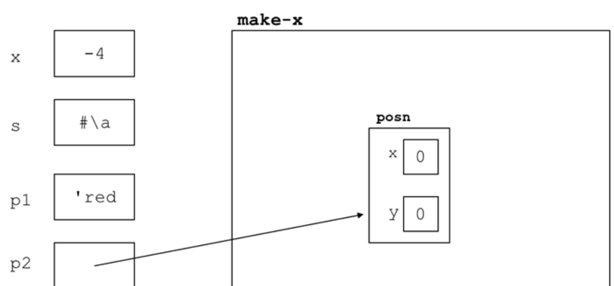
What is happening in memory?

`(set! p1 'red)`



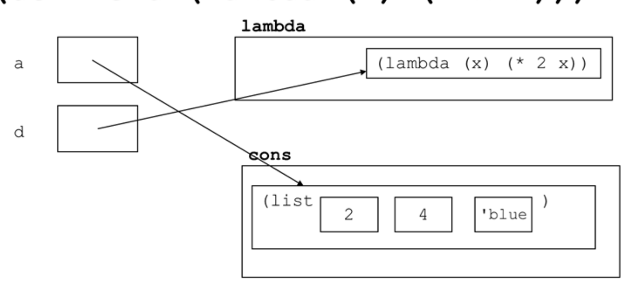
What is happening in memory?

`(set! p2 (make-posn 0 0))`



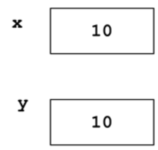
List and Function values

`(define a (list 2 4 'blue))`
`(define d (lambda (x) (* 2 x)))`



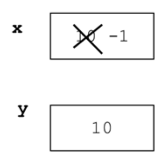
What happens when we use one variable's value to initialize another variable?

```
(define x 10)
(define y x)
```



What if we change one of them?

```
(define x 10)
(define y x)
(set! x -1)
```

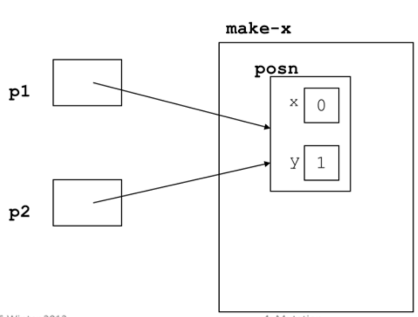


The value of y does not change when **x** is changed. Similarly, the value of **x** does not change if **y** is changed.

(set! v …**)** changes only one box – the one labelled **v**.

How about non-atomic values?

```
(define p1 (make-posn 0 1))
(define p2 p1)
```



Both variables point to the same object in memory.

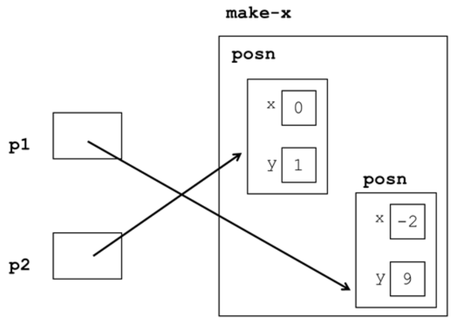
p2 was not initialized with a **make-posn** call, so no new block of memory initialized.

The arrow value was "copied".

p1 and **p2** are called ***aliases***.

And if we change one …

`(set! p1 (make-posn -2 9))`



Only the box labelled **`p1`** changes. **`p2`** still points to the original **`posn`** value.

Again, **`set!`** can only change the value of one variable.

Aliases

- Two variables reference the same object in memory
- Changing the value of one variable does not change the other
- Can be tricky – be careful!
- It is about to become trickier …

Mutating structures

"Automatic" functions for structures in Advanced Student Scheme:

- constructor
- field accessors
- type predicate
- ***NEW*** field mutators

  `(set-struct-field! str expr)`

  Changes the value of indicated field for the variable **`str`** to the value of **`expr`**

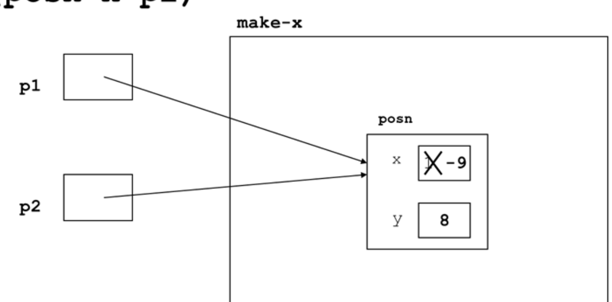
Aliases and Mutating structures

- Suppose: variables **x** and **y** are aliases for a structure
- Action: change the value of a field of **x**
- Effect: changes that field for **y** as well (since there is only one object)

```
(define p1 (make-posn 1 8))
(define p2 p1)
(set-posn-x! p1 -9)
(posn-x p2)
```



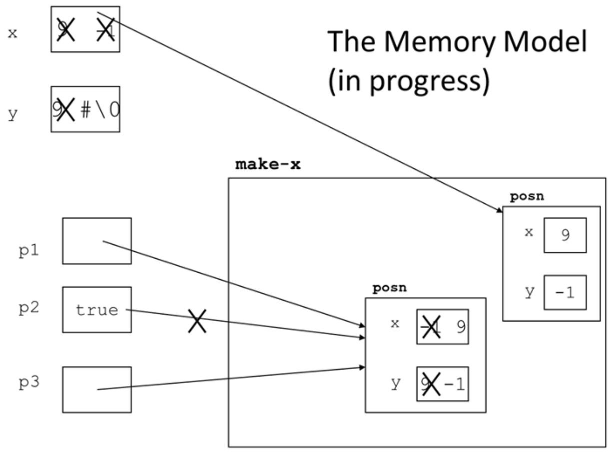
```
(define p1 (make-posn 1 8))
(define p2 p1)
(set-posn-x! p1 -9)
(posn-x p2)
```



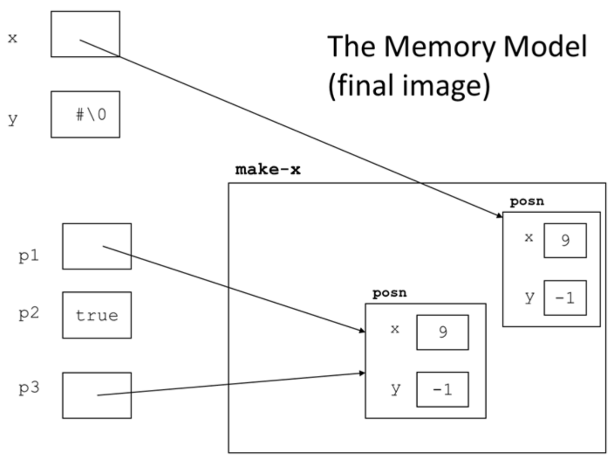
A bigger example to trace

```
(define x 9)
(define y x)
(set! x -1)
(define p1 (make-posn x y))
(define p2 p1)
(define p3 p2)
(set-posn-x! p1 y)
(set-posn-y! p2 x)
(set! y #\0)
(set! p2 true)
(set! x (make-posn (posn-x p1) (posn-y p1)))
```

The Memory Model (in progress)



The Memory Model (final image)



When are objects equal?

Suppose **`x`** and **`y`** are structures of the same type

`(equal? x y)`

- Produces true only if all the fields of **`x`** and **`y`** have equal values

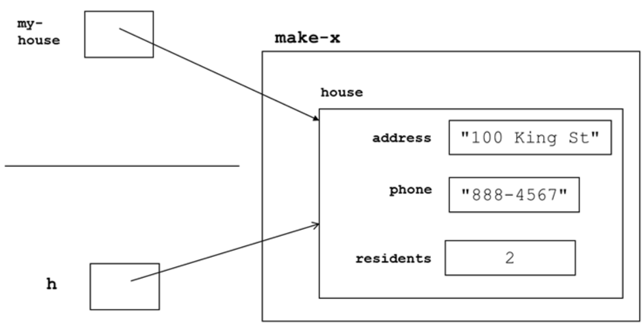
`(eq? x y)`

- Produces true only if **`x`** and **`y`** point to the same value in memory (i.e. only if they are aliases)

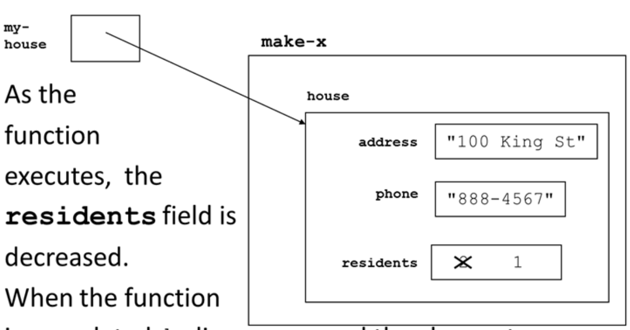
Function parameters and aliases

```
(define-struct house (address phone
  residents))
(define (someone-moves-out h)
  (cond
    [(zero? (house-residents h))
            'alreadyempty]
    [else (set-house-residents!
           h (sub1 (house-residents h)))]))

(define my-house (make-house "100 King St"
  "888-4567" 2))
(someone-moves-out my-house)
```

When function is called, parameter **`h`** and **`my-house`** become aliases.



As the function executes, the **`residents`** field is decreased. When the function is completed, `h` disappears, and the change to **`residents`** is still there.

Aliases with Lists

```
(define L1 (list 2 3 4))
(define L2 L1)
(first L2)
(set! L1 empty)
(rest L2)
```


Example: Online Shopping

Write a Scheme program that simulates on-line shopping by keeping track of all purchases an individual is planning to buy.

- For each purchase, you need the purchase price (before taxes) and the number of items being ordered
- You must remember all the items in a "cart"
- You must remember the total price of all purchases (including taxes)

Getting Started

- Data Analysis:
  - What structures are needed?
  - What state variables are needed?
  - What constants are needed?
- How is mutation involved?

Functions to complete

- **`add-to-cart`**: Consumes the cost of an item and the number of items being ordered, and adds the purchase to the front of the cart. The total owed is also updated.
- **`clear-cart`**: Empty the cart and reset the total owed.
- **`remove-last-item`**: If the cart is not empty, update the cart and the total owed by removing the item at the front of the cart. If the cart is empty, produce "Cart is empty".
- **`check-out`**: Produces the total owed and clears the cart.

Mutation Review

When writing Scheme programs involving mutation, the order of evaluation of mutation expressions is critical.

Value expressions: contain no mutation expressions (everything we've seen before Module 5).

Mutation expressions: change the value of at least one variable or structure field.

Summary: Expressions involving Mutation

1. **`(set! v value-expr)`**
2. **`(set-struct-field! v value-expr)`**

Note: in these mutations, **`value-expr`** should never contain a mutation expression

Summary: Expressions involving Mutation

3. **`(begin ex1 ex2 … exn)`**
   All `ex`'s should be mutation expressions, except possibly the last one, which can be either a mutation expression or a value expression.
4. **`(cond [q1 a1]`**
```
(cond [q1 a1]
      [q2 a2] …
      [qn a3])
```
   where *at least one* **`ai`** is a mutation expression.
   `qi's` should all be value expressions.

Summary: Expressions involving Mutation

5. A function application where the function includes a mutation expression.
   (e.g., **`(define (f x) (set! v x))`**, then the expression **`(f 2)`** is a mutation expression)
6. **`(local [defns] mutation-expr)`**
   Local **`defns`** may be mutation expressions (and their application/use should be as a mutation expression)

Other important information

- The value passed as an argument to a function should never be a mutation expression, don't:
  `(f (set! v e) 3)`
- A mutation expression in the body of a function is evaluated at the time of the application of the function.
- We can write functions that take no parameters:
```
;; my-fcn: (void) -> (void)
(define (my-fcn) (set! x 0))(my-fcn)
```

(void)

- Can appear in three places: n a contract; as the produced value in the Purpose; as an expression

```
;; change-neg-x: posn -> (void)
;; Purpose: Produces (void)
;; Effects: If p1 is (make-posn -3 4), after
;; calling (change-neg-x p1), p1 has been
;; changed to (make-posn 3 4).
(define (change-neg-x p)
  (cond
    [(< (posn-x p) 0) (set-posn-x! p 0)]
    [else (void)]))
```


Goals of Module 4

- Understand basic memory model
- Understand mutation of atomic variables
- Understand mutation of structures
- Understand aliasing and its effects