

M. C. Escher, Day and Night

Class 18: Mutation



CS150: Computer Science
University of Virginia
Computer Science

Greg Humphreys

<http://www.cs.virginia.edu/~humper>

Menu

- Office Hour Changes
- Mutation Primitives
- Programming with Mutation
- PS 5
- Movie Scheduling

Office Hours

- Monday/Tuesday lab hours going away
 - Will still have them this week
- New replacement lab hours
 - Wednesday 3-4:30
 - Saturday 1-4 (not this week -- spring break!)
- In response to overwhelming feedback on survey forms
 - Remember, you don't need to wait until I hand out a form!

From Lecture 3:

Evaluation Rule 2: Names

If the expression is a *name*, it evaluates to the value associated with that name.

```
> (define two 2)
> two
2
```

Names and Places

- A name is not just a value, it is a **place** for storing a value.
- **define** creates a new place, associates a name with that place, and stores a value in that place

```
(define x 3)
```

x: 3

Bang!

set! ("set bang") changes the value associated with a place

```
> (define x 3)
> x
3
> (set! x 7)
> x
7
```

x: 7

Who comes up with this stuff?

!	Bang
#	Hash
*	Splat
~	Twiddle
'	Tick
`	Backtick
#!	Shebang ("Hashbang" acceptable)

set! should make you nervous

```
> (define x 2)
> (nextx)
3           Function application can
> (nextx)   now have side effects!
4           Stuff can change out from
> x         under you!
4
```

Defining nextx

```
(define (nextx)
  (set! x (+ x 1))
  x)
```

syntactic sugar for

```
(define nextx
  (lambda ()
    (begin
      (set! x (+ x 1))
      x))))
```

Evaluation Rules

```
> (define x 3)
> (+ (nextx) x)
7 or 8?
> (+ x (nextx))
9 or 10?
```

DrScheme evaluates application sub-expression left to right, but Scheme evaluation rules allow any order.

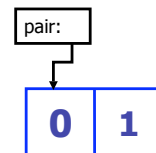
set-car! and set-cdr!

```
(set-car! p v)
Replaces the car of the cons p with v.
```

```
(set-cdr! p v)
Replaces the cdr of the cons p with v.
```

These should scare you even more than set!!

```
> (define pair (cons 1 2))
> pair
(1 . 2)
> (set-car! pair 0)
> (car pair)
0
> (cdr pair)
2
> (set-cdr! pair 1)
> pair
(0 . 1)
```

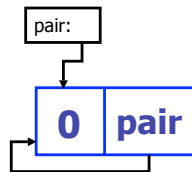


Any reason to be afraid yet?

```

> pair
(0 . 1)
> (set-cdr! pair pair)
> (car pair)
0
> (car (cdr pair))
0
> (car (cdr (cdr pair)))
0
> pair
#0=(0 . #0#)

```



Functional Programming

- Programming without mutation
 - Side-effects like printing and drawing on the screen are really mutations (of the display, printer, bell, etc.)
- If an expression has a value, it is always the same – order of evaluation doesn't matter
- Substitution model of evaluation works fine

Imperative Programming

- Programming with mutation (assignment)
- Value of an expression might be different depending on *when* it is evaluated
- Substitution model of evaluation doesn't work anymore!

Alas, poor Substitution!

```

(define (nextx) (set! x (+ x 1)) x)
> (define x 0)
> ((lambda (x) (+ x x)) (nextx))
2

```

Substitution model:

```

(+ (nextx) (nextx))
(+ (begin (set! x (+ x 1)) x) (begin (set! x (+ x 1)) x))
(+ (begin (set! 0 (+ 0 1)) 0) (begin (set! 0 (+ 0 1)) 0))
(+ 0 0)
0

```

Why would a programming language allow mutation?

- Does it allow us to express computations we couldn't express without it?
 - No!** We can express all computations without mutation. (We'll see why before the end of the course...)
- Mutation allows us to express some computations more naturally and efficiently
 - But it also makes everything else more complicated

map

Functional Solution: A procedure that takes a procedure of one argument and a list, and returns a list of the results

```

(define (map proc lst)
  (if (null? lst) null
      (cons (proc (car lst))
            (map proc (cdr lst)))))

```

Imperative Solution

```
(define (map proc lst)
  (if (null? lst) null
      (cons (proc (car lst))
            (map proc (cdr lst)))))
```

A procedure that takes a procedure and list as arguments, and replaces each element in the list with the value of the procedure applied to that element.

```
(define (map! f lst)
  (if (null? lst) (void)
      (begin
        (set-car! lst (f (car lst)))
        (map! f (cdr lst)))))
```

Programming with Mutation

```
> (map! square (intsto 4))
> (define i4 (intsto 4))
> (map! square i4)
> i4
(1 4 9 16)
```

Imperative

```
> (define i4 (intsto 4))
> (map square i4)
(1 4 9 16)
> i4
(1 2 3 4)
```

Functional

PS5

You've all started, right?

Databases

- Database is tables of fields containing values
- Commands for inserting, selecting and mutating entries in the tables

number	lastname	firstname	university
1	Washington	George	none
2	Adams	John	Harvard
3	Jefferson	Thomas	William and Mary

Representing Tables

- A table is just a cons pair:
 - fields (list of fields)
 - list of entries
 - Each entry is a list of values

number	lastname	firstname	university
1	Washington	George	none
2	Adams	John	Harvard
3	Jefferson	Thomas	William and Mary

```
(define presidents
  (make-table (list 'number 'lastname 'firstname 'university)))
(table-insert! presidents (list 1 "Washington" "George" "none"))
```

Selecting Entries

number	lastname	firstname	university
1	Washington	George	none
2	Adams	John	Harvard
3	Jefferson	Thomas	William and Mary

```
> (table-select presidents `university
  (lambda (item)
    (not (string=? item "Harvard"))))
(('number 'lastname 'firstname 'university)
 .(("Washington" "George" "none")
  ("Jefferson" "Thomas" "William and Mary"))
```

PS5

- You will implement database commands for selecting and deleting entries from a table, and use your database to implement an auction service
- The database commands similar to SQL used by commercial database
- In PS9, you will use SQL to interact with a production quality database (MySQL)

Charge

- PS5
 - Keep on doing it
- Feedback Survey
 - Please hand this in! What I have gotten so far has been very helpful.
 - Mail me if you need another copy
- When can we watch Sneakers?

Sneakers Options

- Tuesday 5 or later
- Wednesday 2 or later
- Thursday 1 or later
- Friday 2 or later