Programming Languages
Scheme part 3
2020

Instructor: Odelia Schwartz

Lots of equalities!

Summary:

- eq?   for symbolic atoms, not numeric (eq? ‘a ‘b)
- =   for numeric, not symbolic   (= 5 7)
- eqv? for numeric and symbolic
- What about equivalence of lists?? Later…

Reminder: Function equalsimp

- For comparing equality between **simple** lists

```
(define (equalsimp lis1 lis2)
  (cond
    ((null? lis1) (null? lis2))
    ((null? lis2) #f)
    ((eq? (car lis1) (car lis2))
          (equalsimp (cdr lis1) (cdr lis2)))
    (else #f)
  )
)
```

Function equal

- What about non simple lists, i.e., **lists within lists**?

Example:

(equal ‘(a (b c)) ‘(a (b c)))

Function equal

- What about non simple lists, i.e., **lists within lists**?

```
(define (equal lis1 lis2)
  (cond
    ((not (list? lis1)) (eq? lis1 lis2))




    (else #f)
  )
)
```

- If lis1 is not a list but rather an atom

Function equal

- What about non simple lists, i.e., **lists within lists**?

```
(define (equal lis1 lis2)
  (cond
    ((not (list? lis1)) (eq? lis1 lis2))

    (else #f)
  )
)
```

Atom comparison

- If lis1 is not a list but rather an atom, return true if first list atom equal to second list atom

Function equal

- What about non simple lists, i.e., **lists within lists**?

```
(define (equal lis1 lis2)
  (cond
    ((not (list? lis1)) (eq? lis1 lis2))
    ((not (list? lis2)) #f)




    (else #f)
  )
)
```

- If lis1 is a list but lis2 is not, return false…

Function equal

- What about non simple lists, i.e., **lists within lists**?

```
(define (equal lis1 lis2)
  (cond
    ((not (list? lis1)) (eq? lis1 lis2))
    ((not (list? lis2)) #f)
    ((null? lis1) (null? lis2))
    ((null? lis2) #f)



    (else #f)
  )
)
```

- If lis1 null then true if lis2 is null, otherwise if lis1 is not null then if lis2 is return false

Function equal

- What about non simple lists, i.e., **lists within lists**?

```
(define (equal lis1 lis2)
  (cond
    ((not (list? lis1)) (eq? lis1 lis2))
    ((not (list? lis2)) #f)
    ((null? lis1) (null? lis2))
    ((null? lis2) #f)



    (else #f)
  )
)
```

- These are all still base cases …

Function equal

- What about non simple lists, i.e., **lists within lists**?

```
(define (equal lis1 lis2)
  (cond
    ((not (list? lis1)) (eq? lis1 lis2))
    ((not (list? lis2)) #f)
    ((null? lis1) (null? lis2))
    ((null? lis2) #f)
    ((equal (car lis1) (car lis2))
         (equal (cdr lis1) (cdr lis2)))
    (else #f)
  )
)
```

Recursive call with car

If recursive with car returns true, then recursion used again on the cdr

- Most interesting part!

Function equal

- What about non simple lists, i.e., **lists within lists**?

```
(define (equal lis1 lis2)
  (cond
    ((not (list? lis1)) (eq? lis1 lis2))
    ((not (list? lis2)) #f)
    ((null? lis1) (null? lis2))
    ((null? lis2) #f)
    ((equal (car lis1) (car lis2))
         (equal (cdr lis1) (cdr lis2)))
    (else #f)
  )
)
```

Recursive call with car

If recursive with car returns true, then recursion used again on the cdr

- **How is this different from simple list function?**

"

equal versus equalsimp

```
(define (equalsimp lis1 lis2)
  (cond
    ((null? lis1) (null? lis2))
    ((null? lis2) #f)
    ((eq? (car lis1) (car lis2))
         (equalsimp (cdr lis1)
(cdr lis2)))
    (else #f)
  )
)
```

```
(define (equal lis1 lis2)
  (cond
    ((not (list? lis1)) (eq? lis1
lis2))
    ((not (list? lis2)) #f)
    ((null? lis1) (null? lis2))
    ((null? lis2) #f)
    ((equal (car lis1) (car
lis2))
         (equal (cdr lis1) (cdr
lis2)))
    (else #f)
  )
)
```

equal versus equalsimp

```
(define (equalsimp lis1 lis2)
  (cond
    ((null? lis1) (null? lis2))
    ((null? lis2) #f)
    ((eq? (car lis1) (car lis2))
         (equalsimp (cdr lis1)
(cdr lis2)))
    (else #f)
  )
)
```

```
(define (equal lis1 lis2)
  (cond
    ((not (list? lis1)) (eq? lis1
lis2))
    ((not (list? lis2)) #f)
    ((null? lis1) (null? lis2))
    ((null? lis2) #f)
    ((equal (car lis1) (car
lis2))
         (equal (cdr lis1) (cdr
lis2)))
    (else #f)
  )
)
```

equal versus equalsimp

- equalsimp

```
((eq? (car lis1) (car lis2))
        (equalsimp (cdr lis1) (cdr lis2)))
```

- equal

```
((equal (car lis1) (car lis2))
        (equal (cdr lis1) (cdr lis2)))
```

equal versus equalsimp

- equalsimp

```
((eq? (car lis1) (car lis2))
       (equalsimp (cdr lis1) (cdr lis2)))
```

- equal

```
((equal (car lis1) (car lis2))
       (equal (cdr lis1) (cdr lis2)))
```

In equal we have recursive calls both for car and cdr; for simple list equal, just needed car for comparison and then just one recursion on cdr

equal

- Function equal we wrote is actually identical to equal? built in function

- Should be used only when necessary, since much slower than other ones we learned

- eq? for symbolic atoms, not numeric (eq? ‘a ‘b)

- = for numeric, not symbolic (= 5 7)

- eqv? for numeric and symbolic

- equal? For lists, including lists within lists

equal

- Function equal we wrote is actually identical to equal? built in function
- Should be used only when necessary, since much slower than other ones we learned

- eq? for symbolic atoms, not numeric (eq? ‘a ‘b)
- = for numeric, not symbolic (= 5 7)
- eqv? for numeric and symbolic
- equal? For lists, including lists within lists

Function equal

- Each of us make a file called equal.scm
- Do (load “equal.scm”) in csi
- Try for some examples

append

- Constructing a new list that contains all elements of two given list arguments

- This is again an actual scheme function

append

- Constructing a new list that contains all elements of two given list arguments
- This is again an actual scheme function

Examples to try:

(append ‘(a b) ‘(c d))

(append ‘((a) b) ‘(c))

(append ‘((a) b) ‘())

append

```
(define (append lis1 lis2)
(cond
 ((null? lis1) lis2)



))
```

- Terminate recursion when first list empty

append

```
(define (append lis1 lis2)
(cond
 ((null? lis1) lis2)
   (else (cons (car lis1)
     (append (cdr lis1) lis2)))
))
```

- What is this doing?

append

```
(define (append lis1 lis2)
(cond
 ((null? lis1) lis2)
   (else (cons (car lis1)
     (append (cdr lis1) lis2)))
))
```

- Repeatedly place elements of first list into second list

append

```
(define (append lis1 lis2)
(cond
 ((null? lis1) lis2)
   (else (cons (car lis1)
     (append (cdr lis1) lis2)))
))
```

- Reminding ourselves of cons (run it on csi):

(cons ‘(a b) ‘(c d))

(cons ‘((a b) c) ‘(d (e f)))

append

```
(define (append lis1 lis2)
(cond
 ((null? lis1) lis2)
   (else (cons (car lis1)
     (append (cdr lis1) lis2)))
))
```

- Reminding ourselves of cons (run it on csi):

(cons ‘(a b) ‘(c d)) returns ((a b) c d)

(cons ‘((a b) c) ‘(d (e f))) returns (((a b) c) d (e f))

Function append

- Each of us make a file called append.scm
- Do (load "append.scm") in csi
- Try for some examples

guess

```
(define (guess lis1 lis2)
        (cond((null? lis1) #f)
        ((member (car lis1) lis2)
                (cons (car lis1) (guess (cdr lis1) lis2)))
        (else (guess (cdr lis1) lis2))
        )
)
```

- What is guess doing?
- Load function into csi and try some examples for two simple lists

guess

```
(define (guess lis1 lis2)
        (cond((null? lis1) #f)
        ((member (car lis1) lis2)
                (cons (car lis1) (guess (cdr lis1) lis2)))
        (else (guess (cdr lis1) lis2))
        )
)
```

- Examples:

(guess '(a b c) '(d e f))

(guess '(a b c) '(d a b))

guess

```
(define (guess lis1 lis2)
        (cond((null? lis1) #f)
        ((member (car lis1) lis2)
                (cons (car lis1) (guess (cdr lis1) lis2)))
        (else (guess (cdr lis1) lis2))
        )
)
```

- Examples:

```
(guess '(a b c) '(d e f))          #f

(guess '(a b c) '(d a b))          (a b)
```

guess

```
(define (guess lis1 lis2)
        (cond((null? lis1) #f)
        ((member (car lis1) lis2)
                (cons (car lis1) (guess (cdr lis1) lis2)))
        (else (guess (cdr lis1) lis2))
        )
)
```

- Examples:

```
(guess '(a (b a) c) '((b a ) c d))

(guess '(a (b c) c) '((b a ) c d))
```

guess

```
(define (guess lis1 lis2)
        (cond((null? lis1) #f)
        ((member (car lis1) lis2)
                (cons (car lis1) (guess (cdr lis1) lis2)))
        (else (guess (cdr lis1) lis2))
        )
)
```

- Examples:

(guess '(a (b a) c) '((b a) c d))   ((b a) c)

(guess '(a (b c) c) '((b a) c d))   (c)

guess

```
(define (guess lis1 lis2)
        (cond((null? lis1) #f)
        ((member (car lis1) lis2)
                (cons (car lis1) (guess (cdr lis1) lis2)))
        (else (guess (cdr lis1) lis2))
        )
)
```

- Yields simple list that contains common elements of its two parameter lists (i.e., intersection of two lists)

let

- Creates **local scope** in which names temporarily bound to values of expressions

let

- Creates **local scope** in which names temporarily bound to values of expressions

- Can be used in evaluation of new expression but cannot be rebound to new values

let

- Creates **local scope** in which names temporarily bound to values of expressions

General form:

```
(let (
   (name-1 expression-1)
   (name-2 expression-2)

                 …

   (name-n expression-n)

   body
))
```

let

- Creates **local scope** in which names temporarily bound to values of expressions

General form:

```
(let (
   (name-1 expression-1)
   (name-2 expression-2)

                ...

   (name-n expression-n)

   body
))
```

Evaluates all the expressions; then temporarily binds the values to the names; evaluates the body

Example: quadratic-roots

- Example:

```
(define (quadratic-roots a b c)
  (let
   (
      (root_part_over_2a (/ (sqrt (- (* b b) (* 4 a c))) (* 2 a)))
      (minus_b_over_2a   (/ (- 0 b) (* 2 a)))
    )
    (list  (+ minus_b_over_2a root_part_over_2a)
           (- minus_b_over_2a root_part_over_2a)
    )
  )
)
```

Example: quadratic-roots

- Each of us make a file called quadratic-roots.scm
- Do (load “quadratic-roots.scm”) in csi
- Try for some examples

Example: quadratic-roots

- Each of us make a file called quadratic-roots.scm
- Do (load "quadratic-roots.scm") in csi
- Try for some examples

Example:
(quadratic-roots 1 2 1)

Example: quadratic-roots

- Each of us make a file called quadratic-roots.scm
- Do (load "quadratic-roots.scm") in csi
- Try for some examples

Example:
(quadratic-roots 1 2 1)   (-1 -1)

Example: quadratic-roots

- Each of us make a file called quadratic-roots.scm
- Do (load "quadratic-roots.scm") in csi
- Try for some examples

It's solving ax^2 + bx + c = 0

And returning a list of the two solutions

Functional forms

- Composition
- Apply-to-all

Functional forms: composition

- Composition
- $h(x) = f(g(x))$

Functional forms: composition

- Composition
- h(x) = f(g(x))

Example:

```
(define (g x) (* 3 x))

(define (f x) (+ 2 x))
```

Functional forms: composition

- Composition
- h(x) = f(g(x))

Example:

```
(define (g x) (* 3 x))

(define (f x) (+ 2 x))
```

One way (we do the math…):

```
(define (h x) (+ 2 (* 3 x)))
```

Functional forms: composition

- Composition
- h(x) = f(g(x))

Example:

```
(define (g x) (* 3 x))

(define (f x) (+ 2 x))
```

One way (we do the math…):

```
(define (h x) (+ 2 (* 3 x)))
```

Do this and run on csi: (h 4)

Functional forms: composition

- Composition
- h(x) = f(g(x))

Example:

```
(define (g x) (* 3 x))

(define (f x) (+ 2 x))
```

One way (we do the math…):

```
(define (h x) (+ 2 (* 3 x)))
```

Do this and run on csi: (h 4) returns 14

Functional forms: composition

- Composition
- h(x) = f(g(x))

```
(define (g x) (* 3 x))

(define (f x) (+ 2 x))
```

Using compose: a built in function:

```
(define (compose f g) (lambda (x) (f (g x ))))

((compose f g) ‘4)

((compose g f) ‘4)
```

Functional forms: composition

- Composition
- h(x) = f(g(x))

```
(define (g x) (* 3 x))

(define (f x) (+ 2 x))
```

Using compose: a built in function:

```
(define (compose f g) (lambda (x) (f (g x ))))

((compose f g) '4)    returns 14

((compose g f) '4)    returns 18
```

Functional forms: composition

- Composition
- h(x) = f(g(x))

```
(define (g x) (* 3 x))

(define (f x) (+ 2 x))
```

And you can build your own but call it compose2

```
(define (compose2 f g) (lambda (x) (f (g x ))))

((compose2 f g) ‘4)    returns 14

((compose2 g f) ‘4)    returns 18
```

Functional forms: apply-to-all

- map function (also built in)

Functional forms: apply-to-all

- map function (also built in). We'll call it mapcar

```
(define (mapcar fun lis)
  (cond
    ((null? lis) '())
    (else (cons (fun (car lis)) (mapcar fun
(cdr lis))))
))
```

Example: quadratic-roots

- Each of us make a file called mapcar.scm
- Do (load “mapcar.scm”) in csi
- Try for some examples

```
(mapcar (lambda (num) (* num num num)) ‘(3 4 2 6))
```

Example: quadratic-roots

- Each of us make a file called mapcar.scm
- Do (load "mapcar.scm") in csi
- Try for some examples

(mapcar (lambda (num) (* num num num)) '(3 4 2 6))

Returns (27 64 8 216)

Adding a list of numbers

- This works: (+ 3 7 10 2)
- This doesn’t work: (+ (3 7 10 2))

Adding a list of numbers

- This works: (+ 3 7 10 2)
- This doesn’t work: (+ (3 7 10 2))

How would we achieve the second option?

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

We'll do a little "trick" …

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

- cons creates new list with + and a_list

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

- cons creates new list with + and a_list

- Why the quote on `+?

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

- cons creates new list with + and a_list
- Why the quote on `+?
- Quote so that eval will not evaluate in evaluation of cons

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

- Adder (+ 1 2 3 4)
- Calls (eval (+ 1 2 3 4))
- And returns (+ 1 2 3 4)

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

- Create adder function and load into csi
- Run on sci adder (+ 1 2 3 4)
- Run on sci (eval (+ 1 2 3 4))

Adding a list of numbers

- We want: (+ (3 7 10 2))

```
(define (adder a_list)
   (cond
   ((null? a_list) 0)
   (else (eval(cons '+ a_list)))
 )
)
```

Examples:

(adder '(1 2 3))

Adding a list of numbers

- We want: (+ (3 7 10 2))

Let’s each write another way of doing this…

Create adder2 function and load into csi

Run on sci (adder2 ‘(3 7 10 2))

Adding a list of numbers

- We want: (+ (3 7 10 2))

Let’s each write another way of doing this…
Hint: use car and cdr

Create adder2 function and load into csi

Run on sci (adder2 ‘(3 7 10 2))