Programming Languages
Scheme part 2

2020

Instructor: Odelia Schwartz

Lots of equalities!

- Predicate function: eq?
- Takes two expressions as parameters
- Returns #t if both parameters are atoms and the two are the same

Lots of equalities!

- Predicate function: eq?
- Takes two expressions as parameters
- Returns #t if both parameters are atoms and the two are the same

Examples: try in csi:

(eq? ‘a ‘a)

(eq? ‘a ‘b)

(eq? ‘a ‘(a b))

Lots of equalities!

- Predicate function: eq?
- Takes two expressions as parameters
- Returns #t if both parameters are atoms and the two are the same

Examples: try in csi:

(eq? ‘a ‘a) returns #t

(eq? ‘a ‘b) returns #t

(eq? ‘a ‘(a b)) returns #f

Lots of equalities!

- Predicate function: eq?
- Takes two expressions as parameters
- Returns #t if both parameters are atoms and the two are the same

Examples: try in csi:

(eq? ‘(a b) ‘(a b))   returns ?

Lots of equalities!

- Predicate function: eq?

Examples: try in csi:

(eq? '(a b) '(a b))   returns ?

- #f in my csi

Lots of equalities!

- Predicate function: eq?

  Examples: try in csi:

  (eq? ‘(a b) ‘(a b))   returns ?

- #f in my csi

- Textbook: returns #f or #t

Lots of equalities!

- Predicate function: eq?

  Examples: try in csi:

  (eq? `(a b) `(a b)) returns ?

- #f in my csi
- Textbook: returns #f or #t
- Because two lists exactly the same often not duplicated in memory; if so pointer to the same list and would return #t; sometimes hard to detect presence of identical list and new one in memory

Lots of equalities!

- Predicate function: eq?

  Examples: try in csi:

  (eq? ‘3.4 ‘(+ 3 .4)) returns ?

Lots of equalities!

- Predicate function: eq?

  Examples: try in csi:

  (eq? ‘3.4 ‘(+ 3 .4))   returns ?

- #f in my csi

- Textbook: returns #f or #t

Lots of equalities!

- Predicate function: eq?

  Examples: try in csi:

  (eq? `3.4 `(+ 3 .4)) returns ?

- #f in my csi
- Textbook: returns #f or #t
- If addition produces new values then not equal to 3.4 #f; or if recognizes already has value will use pointer to old 3.4 and return #t

"

Lots of equalities!

- function =
  - Works for numeric but not symbolic

Lots of equalities!

- function =
  - Works for numeric but not symbolic
  - Works as you would expect for numeric

- (= 2 3)
- (= 2.0 2.0)
- (= 3.4 (+ 3 .4))
- (= 3 3.0)

Lots of equalities!

- function eqv?
  - Works for symbolic and numeric …

Lots of equalities!

- function eqv?
    - Works for symbolic and numeric …

Try these:

- (eqv? ‘a ‘a)
- (eqv? ‘a ‘b)
- (eqv? 3 3)
- (eqv? ‘a 3)

Lots of equalities!

- function eqv?
  - Works for symbolic and numeric …

Try these:

- (eqv? ‘a ‘a) #t
- (eqv? ‘a ‘b) #f
- (eqv? 3 3) #t
- (eqv? ‘a 3) #f

Lots of equalities!

- function eqv?
    - Works for symbolic and numeric …

Try these:

- (eqv? 3.4 (+ 3 .4))
- (eqv? 3.0 3)

Lots of equalities!

- function eqv?
    - Works for symbolic and numeric …

Try these:

- (eqv? 3.4 (+ 3 .4)) #t
- (eqv? 3.0 3) #f may depend on interpreter

Lots of equalities!

Summary:

- eq? for symbolic atoms, not numeric (eq? ‘a ‘b)
- = for numeric, not symbolic (= 5 7)
- eqv? for numeric and symbolic

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…

More predicates

- list?

More predicates

- list? Returns #t if its single argument is a list and false otherwise

More predicates

- list?   Returns #t if its single argument is a list and false otherwise

Try some examples with csi:

- (list? ‘(x y))
- (list? ‘x)
- (list? ‘(5))
- (list? ‘((a) b c))
- (list? 5)

More predicates

- list? Returns #t if its single argument is a list and false otherwise

Try some examples with csi:

- (list? ‘(x y)) #t
- (list? ‘x) #f
- (list? ‘(5)) #t
- (list? ‘((a) b c)) #t
- (list? 5) #f

More predicates

- list? versus list

Try some examples with csi:

- (list? ‘(5))
- (list ‘(5))

More predicates

- list? versus list

Try some examples with csi:

- (list? ‘(5)) #t
- (list ‘(5)) ((5))

More predicates

- null?
- Returns #t if the empty list

More predicates

- null?
- Returns #t if the empty list

Examples: try them…

(null? ‘(a b))

(null? ‘())

(null? ‘(()))

More predicates

- null?
- Returns #t if the empty list

Examples: try them…

(null? `(a b)) #f

(null? `()) #t

(null? `(())) #f

Writing example scheme functions

- Will use recursion instead of iteration…

Function member

- Note: scheme has an actual function named member
- Goal: is an atom a member of a list?

Function member

- Goal: is an atom a member of a list?
- (member '1 '(1 2 3)) evaluates #t
- (member 'a '(b b c)) evaluates #f

Function member

```
(define (member atm lis)



)
```

Function member

```
(define (member atm lis)
  (cond



  )
)
```

Function member

```
(define (member atm lis)
  (cond
    ((null? lis) #f)



  )
)
```

- Try in csi: (null? '())

Function member

```
(define (member atm lis)
  (cond
    ((null? lis) #f)
    ((eq? atm (car lis)) #t)

  )
)
```

- null must come first because (car ‘()) is an error
- try (eq? ‘a (car ‘(a b c)))

Function member

```
(define (member atm lis)
  (cond
    ((null? lis) #f)
    ((eq? atm (car lis)) #t)
    (else (member atm (cdr lis)))
  )
)
```

- else recurse on the remainder of the list

Function member

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

Function equalsimp

- For comparing equality between simple lists
- A simple list is a list that does not have lists within it

Function equalsimp

- For comparing equality between simple lists
- A simple list is a list that does not have lists within it
- ‘(1 2 3) is a simple list
- ‘(1 (2) 3) is not a simple list

Function equalsimp

- For comparing equality between simple lists
- ‘(1 2 3) ‘(1 2 3) are equal simple lists
- ‘() ‘() are equal simple lists

Function equalsimp

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





  )
)
```

- If first list empty, second list also checked to see if empty; if so returns #t

Function equalsimp

```
(define (equalsimp lis1 lis2)
  (cond
    ((null? lis1) (null? lis2))
    ((null? lis2) #?)



  )
)
```

- Second list empty when first is not
- What should we return in this case?

Function equalsimp

```
(define (equalsimp lis1 lis2)
  (cond
    ((null? lis1) (null? lis2))
    ((null? lis2) #f)




  )
)
```

- Second list empty when first is not
- So return false

Function equalsimp

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

    )
  )
)
```

- If first element of each list equal, what should we do?

Function equalsimp

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

  )
)
```

- If first element of each list equal, recurse on remaining of each list

Function 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)
  )
)
```

- If above failed, i.e. first element of each list not equal to each other, then no need to continue; returns false…

Function equalsimp

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

Find the kth element of a list

- Breakout groups (approx. 20 minutes?)
- Write function and try
  ```
  (findk '(1 4 6 8) 1)
  (findk '(1 4) 2)
  (findk '(1 4 6 8) 3)
  (findk '(1 4 6 8) 0)
  (findk '(1 4 6 8) 5)
  ```

Find the kth element of a list

- Breakout groups

- Write function findk and try
  (findk '(1 4 6 8) 1)
  (findk '(1 4) 2)
  (findk '(1 4 6 8) 3)
  (findk '(1 4 6 8) 0)
  (findk '(1 4 6 8) 5)

Hint: you can use recursion and an if function within a cond

Find the kth element of a list

- Breakout groups

- Write function findk and try

```
(findk '(1 4 6 8) 1)
(findk '(1 4) 2)
(findk '(1 4 6 8) 3)
(findk '(1 4 6 8) 0)
(findk '(1 4 6 8) 5)
```

Hint: you can use recursion and an if function within a cond

- Reminder if format:

```
(define (themax a b)
   (if (> a b)
     a
     b))
```

Find the kth element of a list

- Breakout groups discussion in main classroom

Find the kth element of a list

My solution:

Discuss…

Find the kth element of a list

- Examples:

```
(findk '(1 4 6 8) 1)
(findk '(1 4) 2)
(findk '(1 4 6 8) 3)
(findk '(1 4 6 8) 0)
(findk '(1 4 6 8) 5)
```

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