CS 131
PROGRAMMING LANGUAGES
(WEEK 7)

UCLA WINTER 2019

TA: SHRUTI SHARAN

DISCUSSION SECTION: 1D

ADMINISTRATIVE INTRODUCTION

TA: Shruti Sharan

Email: shruti5596@g.ucla.edu
(Will reply by EOD)

Office Hours: Mondays 1.30PM – 3.30PM
Location: Eng. VI 3rd Floor

Discussion Section: Friday 4.00-5.50PM
Location: 2214 Public Affairs

ADMINISTRATION

- Midterms are up on Gradescope.
  - Meet the respective TAs during Office hours for clarifications/regrade
- HW4 is due tonight by 11:55PM
- HW5: Due Friday, February 29th.
  - not compiling → no credit
  - code should behave exactly according to spec
  - check Piazza for clarifications

TODAY'S AGENDA

SCHEME: INTRODUCTION

OPERATORS IN SCHEME

LIST PROCESSING

HOMEWORK #5

SCHEME

SCHEME - INSTALLATION

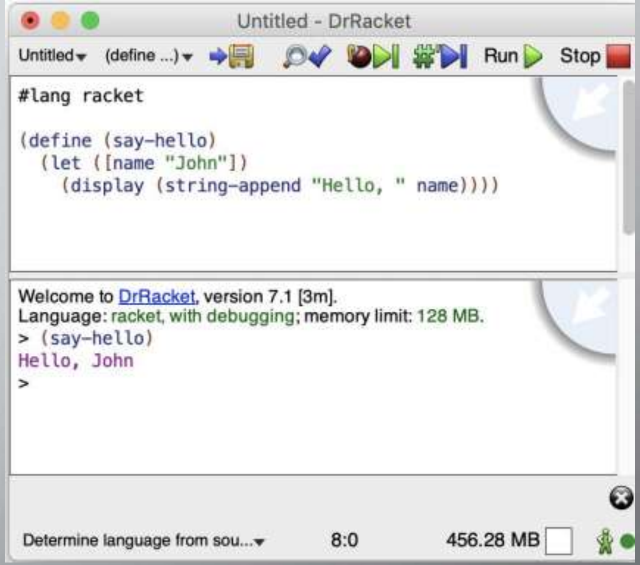
- For the homework, we will use Racket, which is a descendant of Scheme
  - Racket implements Scheme standard plus some additional features
- http://download.racket-lang.org
  - Choose your OS and 32 or 64 bit version
- Racket 7.1 version
  - We will be using this release to check your HW
  - If running on Seasnet be sure to use this version
- You can use DrRacket IDE or any text editor
  - DrRacket is a very minimal IDE, just a text editor and an interactive environment
  - DrRacket might make your life a lot easier...

SCHEME - BASICS

- Functional Language
- Part of LISP language family.
  - LISP invented in 1958 by John McCarthy
  - Introduced many new concepts, for example:
    - Garbage collection
    - Program code as a data structure
- Dialect of Lisp
  - Created 1970s at MIT AI Lab
  - Historically very popular language in academia
  - You’ll encounter Scheme again in CS161 - Artificial Intelligence!
- Minimalist design
- Static scoping but Dynamic Typing

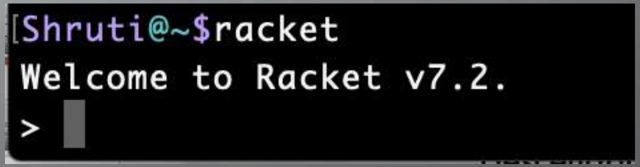
WHAT IS RACKET?

- Racket is a programing language
  - Dialect of Lisp
  - Descendant of Scheme
- Also a family of programming languages
  - It includes all the variants of Racket
- Main tools are:
  - racket: compiler, interpreter, run-time system.
  - DrRacket: programming environment
  - raco: command line tool to install racket packages, build binaries



```
#lang racket

(define (say-hello)
  (let ([name "John"])
    (display (string-append "Hello, " name))))
```

```
Welcome to DrRacket, version 7.1 [3m].
Language: racket, with debugging; memory limit: 128 MB.
> (say-hello)
Hello, John
>
```



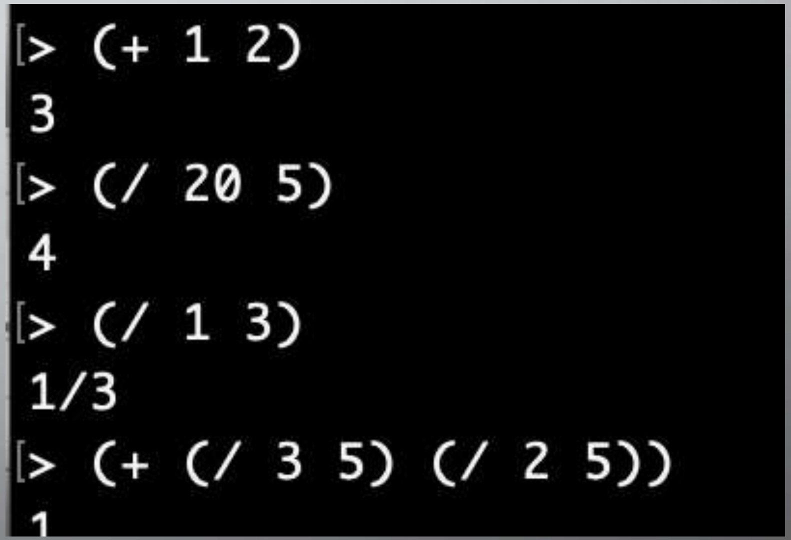
```
Shruti@~$racket
Welcome to Racket v7.2.
>
```

BASIC SYNTAX

- Comments
  - ; (semi-colon) starts a line comment
  - #| Block comment |#
- Numbers
  - 1, 1/2, 3.14, 6.02e+23
- Strings
  - "Hello, World!"
- Booleans
  - #t, #f

ARITHMETIC OPERATORS

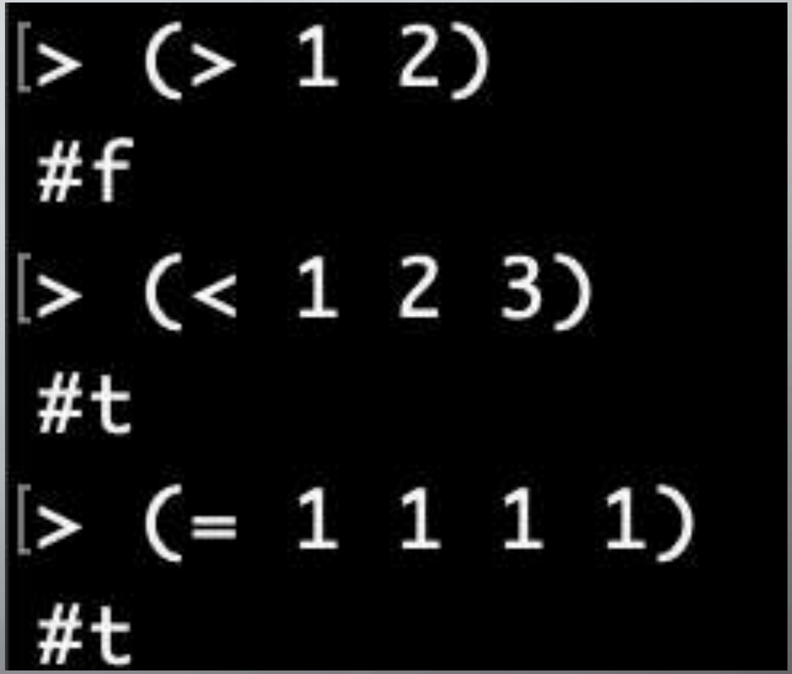
- Procedures use prefix operators
- + , - , * , /



```
[> (+ 1 2)
 3
[> (/ 20 5)
 4
[> (/ 1 3)
 1/3
[> (+ (/ 3 5) (/ 2 5))
 1
```

COMPARISON OPERATORS

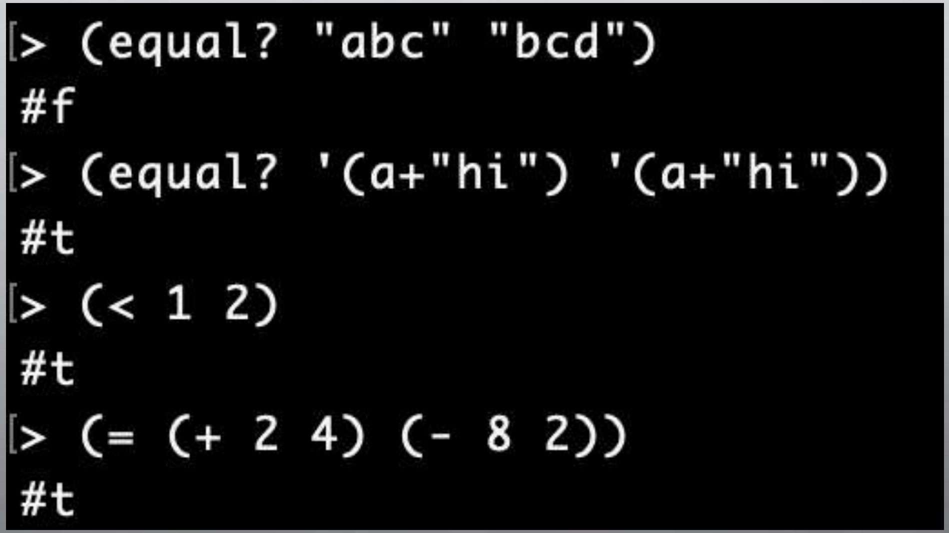
- Basic comparison operators
  - (=, >,<, <=, >=)



```
[> (> 1 2)
 #f
[> (< 1 2 3)
 #t
[> (= 1 1 1 1)
 #t
```

BINARY/ BOOLEAN VALUES

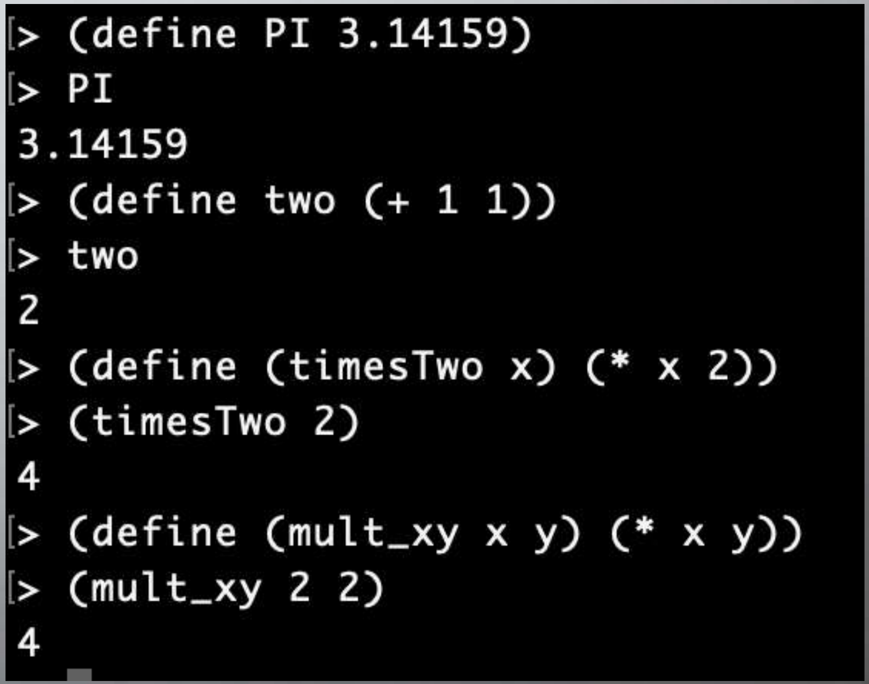
- Represented with #t (true) and #f (false)
- Anything other than #f is interpreted as true



```
[> (equal? "abc" "bcd")
#f
[> (equal? '(a+"hi") '(a+"hi"))
#t
[> (< 1 2)
#t
[> (= (+ 2 4) (- 8 2))
#t
```

DEFINITIONS

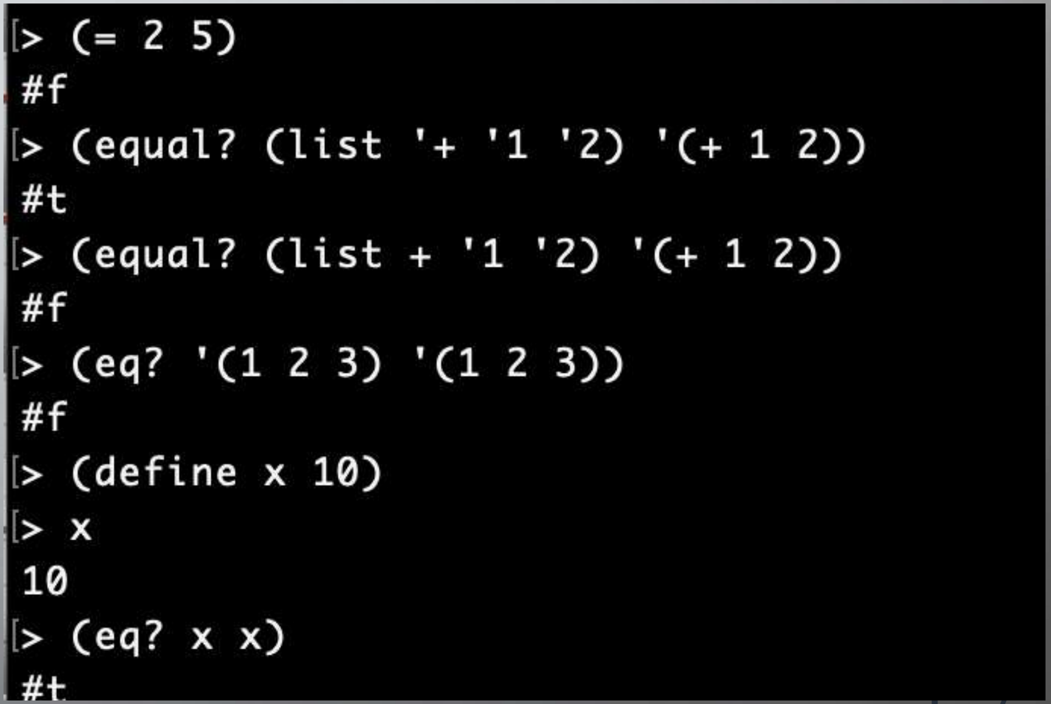
- (define <id> <expr>)
  - Defines a function that returns an expression
  - Defines a variable
- (define (<id> <id>*) <expr>+)
  - Defines a function with 0 or more arguments
    - First <id> is function name rest are arguments.
    - <expr> defines body of the function.
    - Function returns result of the last <expr>



```
> (define PI 3.14159)
> PI
3.14159
> (define two (+ 1 1))
> two
2
> (define (timesTwo x) (* x 2))
> (timesTwo 2)
4
> (define (mult_xy x y) (* x y))
> (mult_xy 2 2)
4
```

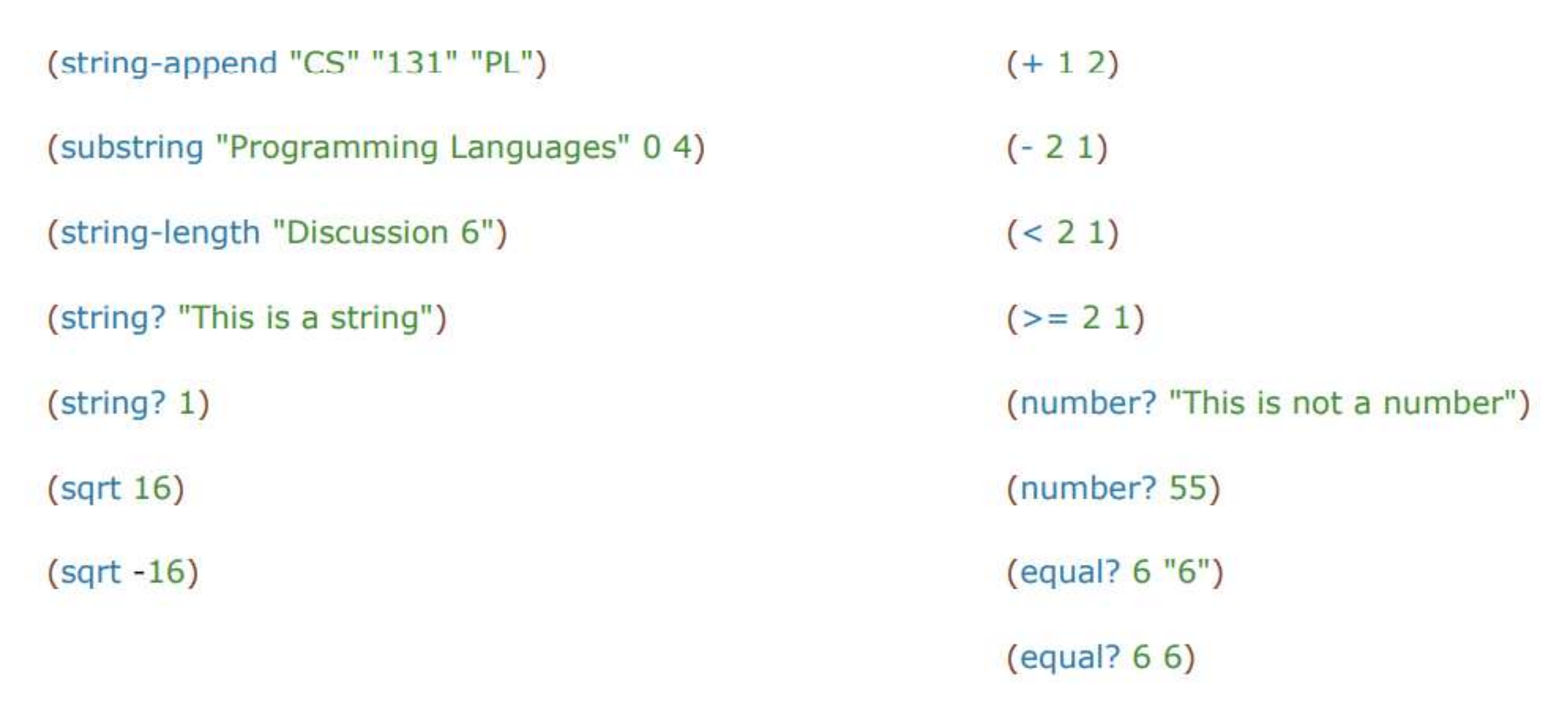
EQUALITY

- **=**
  - Tests the equivalency of two numbers
- **equal?**
  - Tests structural equivalence of two items (lists, vectors, etc.)
- **eq?**
  - Tests whether two items refer to the same thing in memory



```
[> (= 2 5)
#f
[> (equal? (list '+ '1 '2) '(+ 1 2))
#t
[> (equal? (list + '1 '2) '(+ 1 2))
#f
[> (eq? '(1 2 3) '(1 2 3))
#f
[> (define x 10)
[> x
10
[> (eq? x x)
#t
```

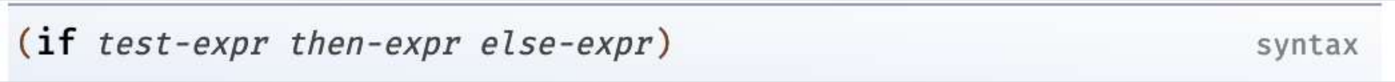
FUNCTION CALLS: COMMON FUNCTIONS



```
(string-append "CS" "131" "PL")

(substring "Programming Languages" 0 4)

(string-length "Discussion 6")

(string? "This is a string")

(string? 1)

(sqrt 16)

(sqrt -16)
```

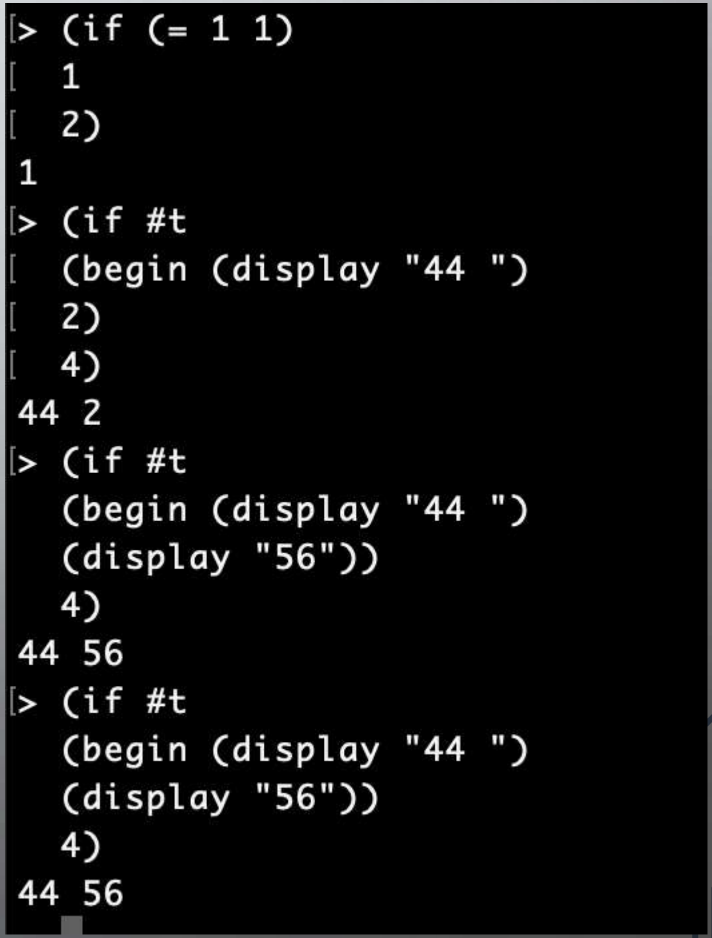
```
(+ 1 2)

(- 2 1)

(< 2 1)

(>= 2 1)

(number? "This is not a number")

(number? 55)

(equal? 6 "6")

(equal? 6 6)
```

CONDITIONALS - if



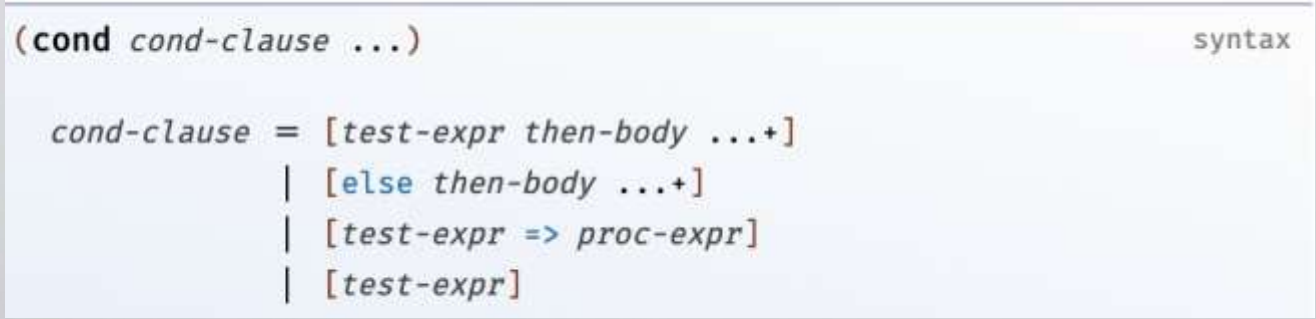
```
(if test-expr then-expr else-expr)                    syntax
```

- Evaluates *test-expr*.
- If it produces #t, then *then-expr* is evaluated, and its results are the result for the if form.
- Otherwise, *else-expr* is evaluated, and its results are the result for the if form.
- Each branch contains a single expression
  - Use begin to execute more than one expression



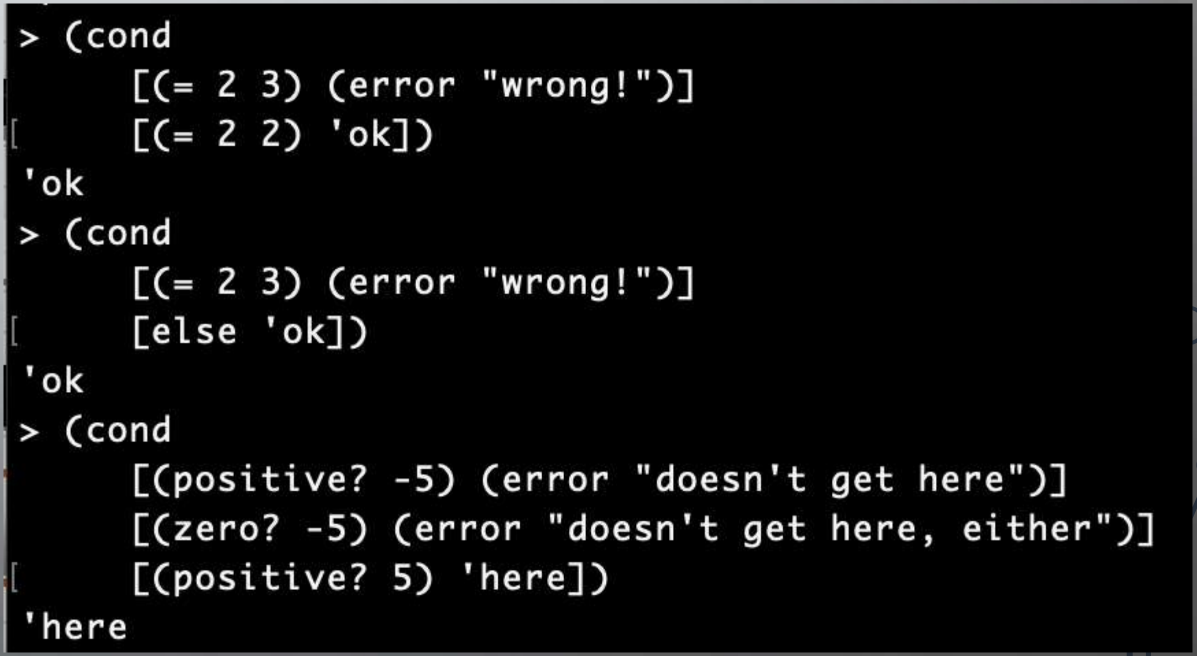
```
[> (if (= 1 1)
[   1
[   2)
1
[> (if #t
[   (begin (display "44 ")
[   2)
[   4)
44 2
[> (if #t
    (begin (display "44 ")
    (display "56"))
    4)
44 56
[> (if #t
    (begin (display "44 ")
    (display "56"))
    4)
44 56
```

CONDITIONALS- cond



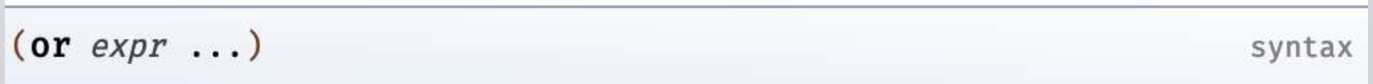
```
(cond cond-clause ...)                                    syntax

  cond-clause = [test-expr then-body ...+]
              | [else then-body ...+]
              | [test-expr => proc-expr]
              | [test-expr]
```

- Cond supports any number of condition branches, and an optional else branch.
- evaluates the condition on the left side of each branch, and stops at the first one that evaluates as true (precisely, the first one that's not #f).
- Then it evaluates the right side of the branch.
- If no branches match, you get <#void>



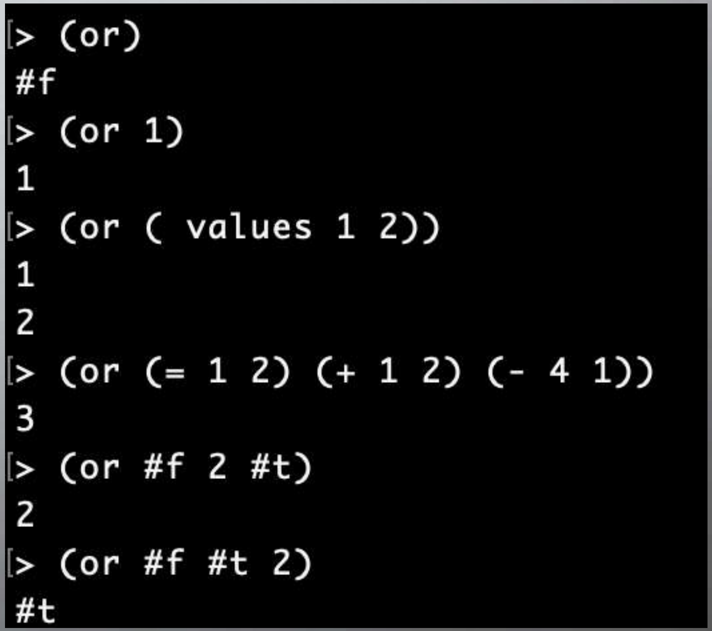
```
> (cond
    [(= 2 3) (error "wrong!")]
    [(= 2 2) 'ok])
'ok
> (cond
    [(= 2 3) (error "wrong!")]
    [else 'ok])
'ok
> (cond
    [(positive? -5) (error "doesn't get here")]
    [(zero? -5) (error "doesn't get here, either")]
    [(positive? 5) 'here])
'here
```

CONDITONALS – or



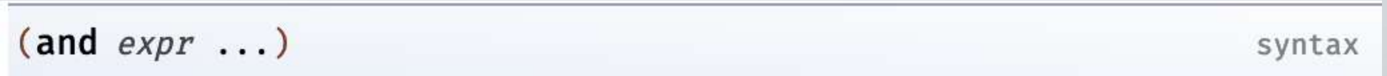
```
(or expr ...)                                                syntax
```

- The first *expr* is evaluated.
- If it produces a value other than #f, that result is the result of the or expression
- Executes every instruction until it has evaluated an expression
- Returns the last thing evaluated
- If no *exprs* are provided, then result is #f.
- Uses **short-circuit evaluation.**



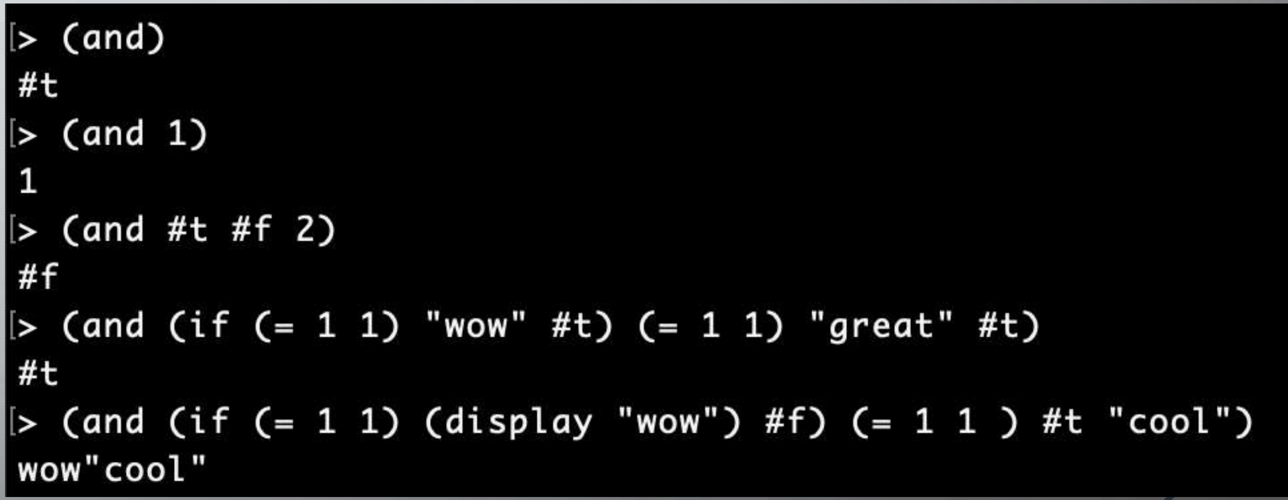
```
> (or)
#f
> (or 1)
1
> (or ( values 1 2))
1
2
> (or (= 1 2) (+ 1 2) (- 4 1))
3
> (or #f 2 #t)
2
> (or #f #t 2)
#t
```

CONDITONALS – and



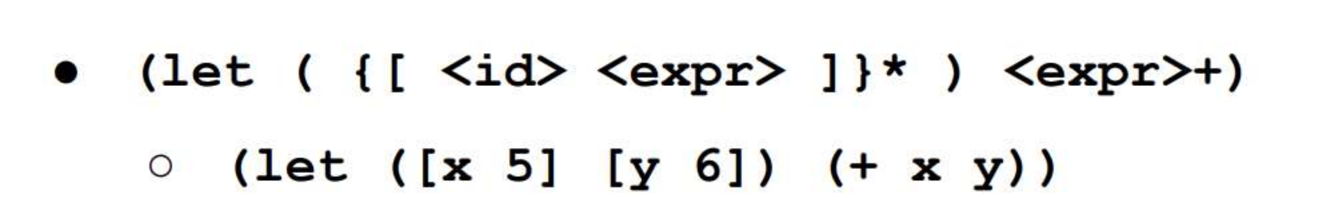
```
(and expr ...)                                                    syntax
```

- The first *expr* is evaluated.
- If it produces #f, the result of the and expression is #f.
- Keeps evaluating all the expressions till all are #t.
- If no *exprs* are provided, then result is #t.



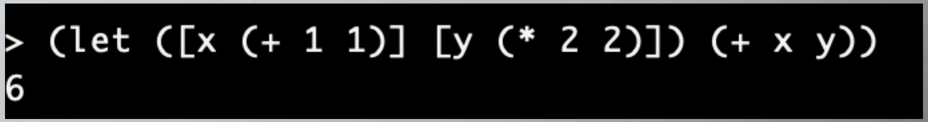
```
[> (and)
#t
[> (and 1)
1
[> (and #t #f 2)
#f
[> (and (if (= 1 1) "wow" #t) (= 1 1) "great" #t)
#t
[> (and (if (= 1 1) (display "wow") #f) (= 1 1 ) #t "cool")
wow"cool"
```

LET AND SCOPING



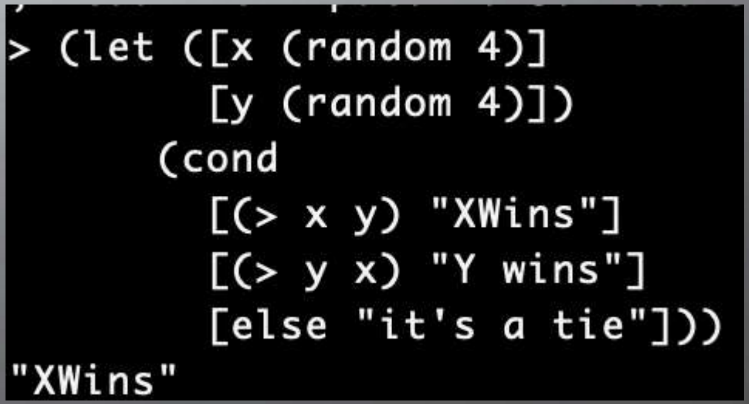
- (let ({[<id> <expr>]}*) <expr>+)
  - ○ (let ([x 5] [y 6]) (+ x y))

- Let :
  - Used to create local bindings.
  - The bindings of let are only available in the body of let but not in the clauses.
    - Use let* for that



```
> (let ([x (+ 1 1)] [y (* 2 2)]) (+ x y))
6
```

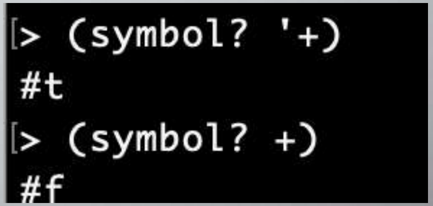
- Scope:
  - Where a variable can be used
  - Scheme is Lexically Scoped



```
> (let ([x (random 4)]
        [y (random 4)])
     (cond
       [(> x y) "XWins"]
       [(> y x) "Y wins"]
       [else "it's a tie"]))
"XWins"
```

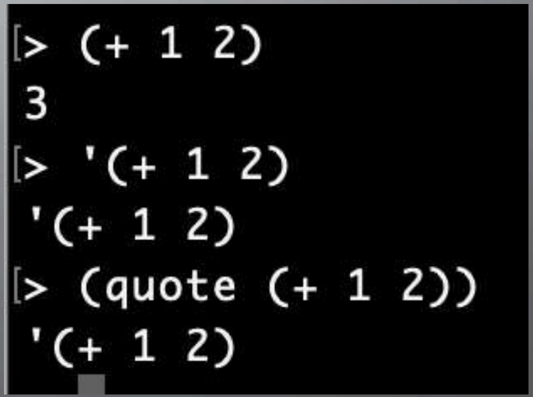
QUOTES

- Suppose I want to use + or equal as a symbol but not as a procedure
- Use (quote +) or ‘+ as shorthand
- Single quote ‘ or quote denotes “treat this as data”



```
[> (symbol? '+)
#t
[> (symbol? +)
#f
```

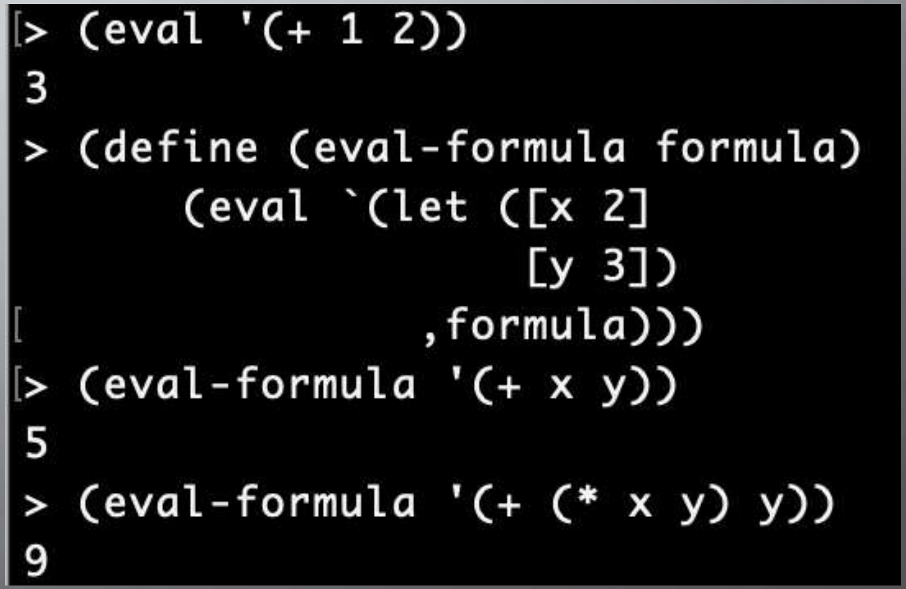
- Quotes evaluates the expression as a data.



```
[> (+ 1 2)
3
[> '(+ 1 2)
'(+ 1 2)
[> (quote (+ 1 2))
'(+ 1 2)
```

EVAL

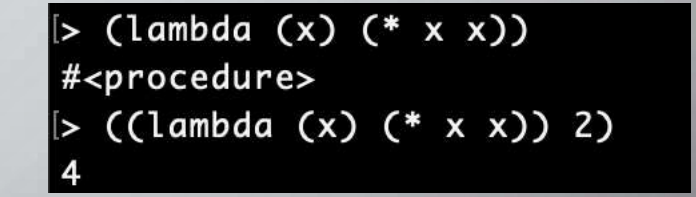
- The eval function takes a representation of an expression or definition (as a “quoted” form) and evaluates it.
- Negates quote
- Takes a list and treats it like a program



```
> (eval '(+ 1 2))
3
> (define (eval-formula formula)
      (eval `(let ([x 2]
                   [y 3])
               ,formula)))
> (eval-formula '(+ x y))
5
> (eval-formula '(+ (* x y) y))
9
```

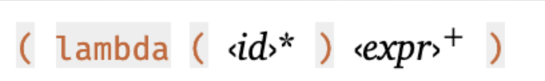
ANONYMOUS FUNCTIONS

- In Racket, you can use a lambda expression to produce a function directly



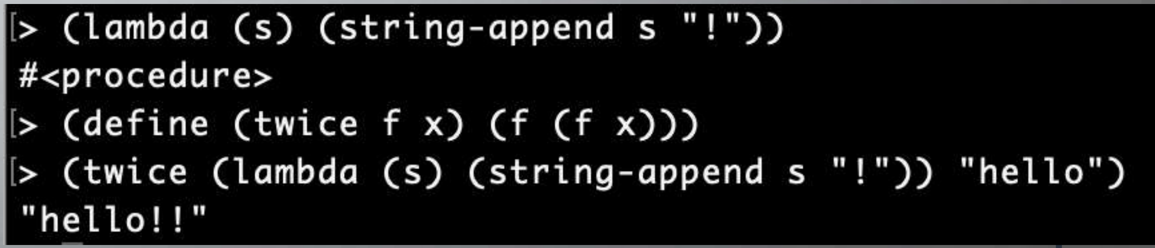
```
[> (lambda (x) (* x x))
#<procedure>
[> ((lambda (x) (* x x)) 2)
4
```

- Syntax:



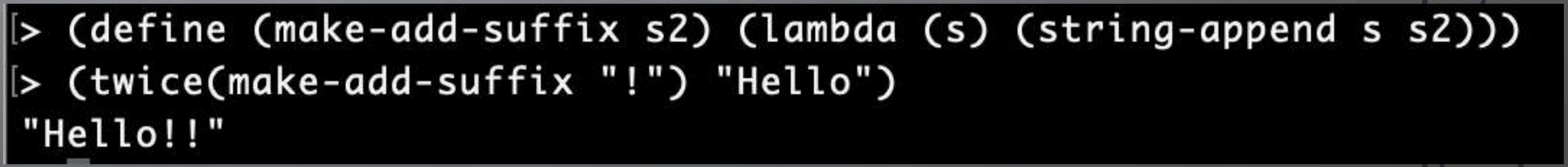
```
( lambda ( ‹id›* ) ‹expr›+ )
```

- Lambda by itself returns a procedure.
  - Does nothing.
  - Same as calling a function without arguments.



```
[> (lambda (s) (string-append s "!"))
#<procedure>
[> (define (twice f x) (f (f x)))
[> (twice (lambda (s) (string-append s "!")) "hello")
"hello!!"
```

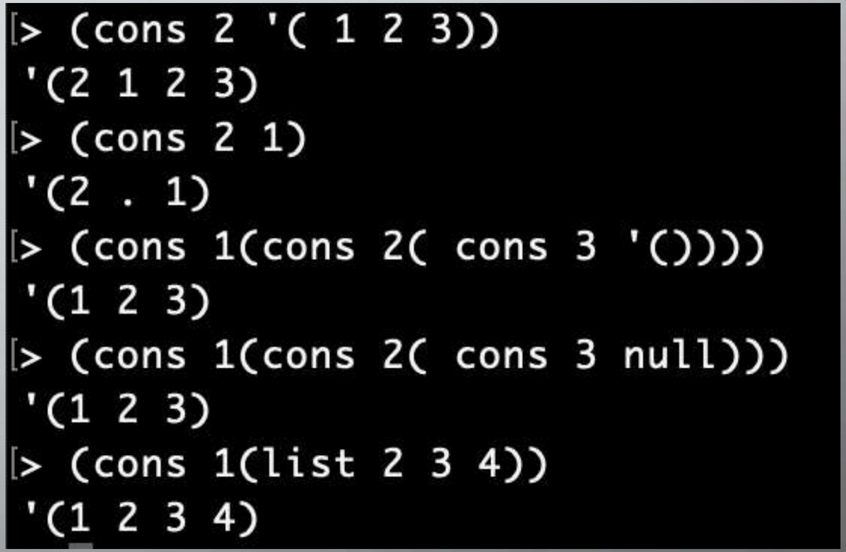
- Lambda can also be returned as a result of a function



```
[> (define (make-add-suffix s2) (lambda (s) (string-append s s2)))
[> (twice(make-add-suffix "!") "Hello")
"Hello!!"
```

cons

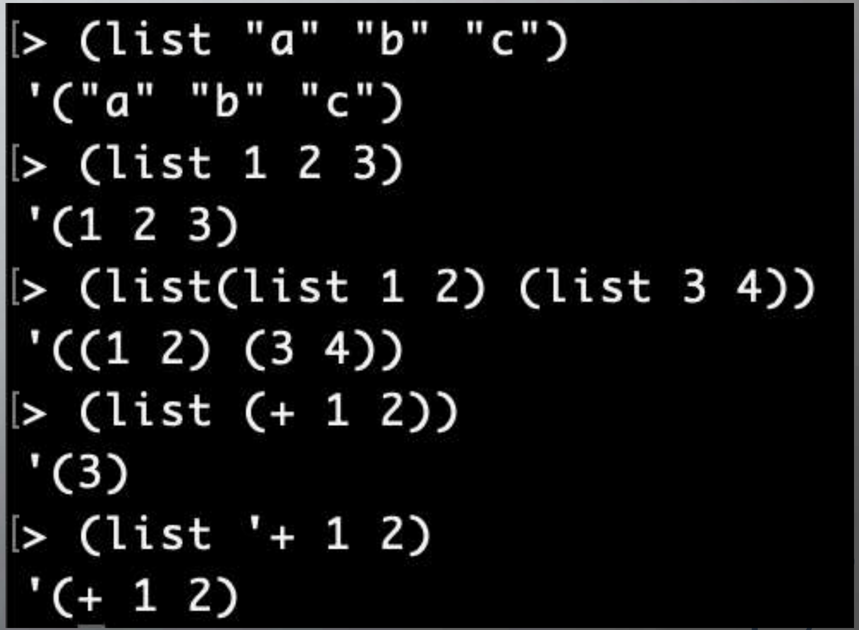
- In the general case it is used for pairs.
- Used to construct a list
- Lists end with a null. ().
- Can return a list or a pair (Improper lists)
- Improper Lists: Don’t end with ‘( )



```
[> (cons 2 '( 1 2 3))
 '(2 1 2 3)
[> (cons 2 1)
 '(2 . 1)
[> (cons 1(cons 2( cons 3 '())))
 '(1 2 3)
[> (cons 1(cons 2( cons 3 null)))
 '(1 2 3)
[> (cons 1(list 2 3 4))
 '(1 2 3 4)
```

LISTS

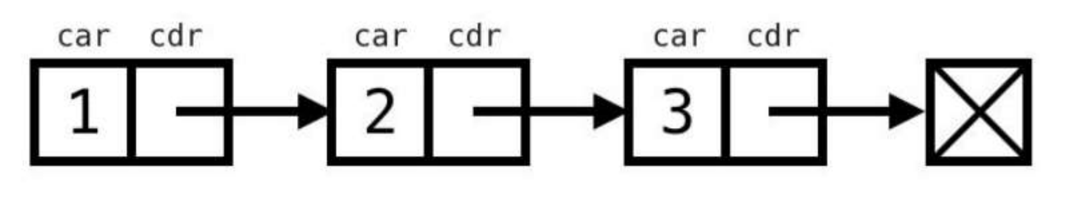
- Racket inherits much of its style from the language Lisp, whose name originally stood for “LISt Processor”.
- Similar to OCaml, but can contain any type .
- The list function takes any number of arguments and returns a list containing the given values.



```
[> (list "a" "b" "c")
 '("a" "b" "c")
[> (list 1 2 3)
 '(1 2 3)
[> (list(list 1 2) (list 3 4))
 '((1 2) (3 4))
[> (list (+ 1 2))
 '(3)
[> (list '+ 1 2)
 '(+ 1 2)
```

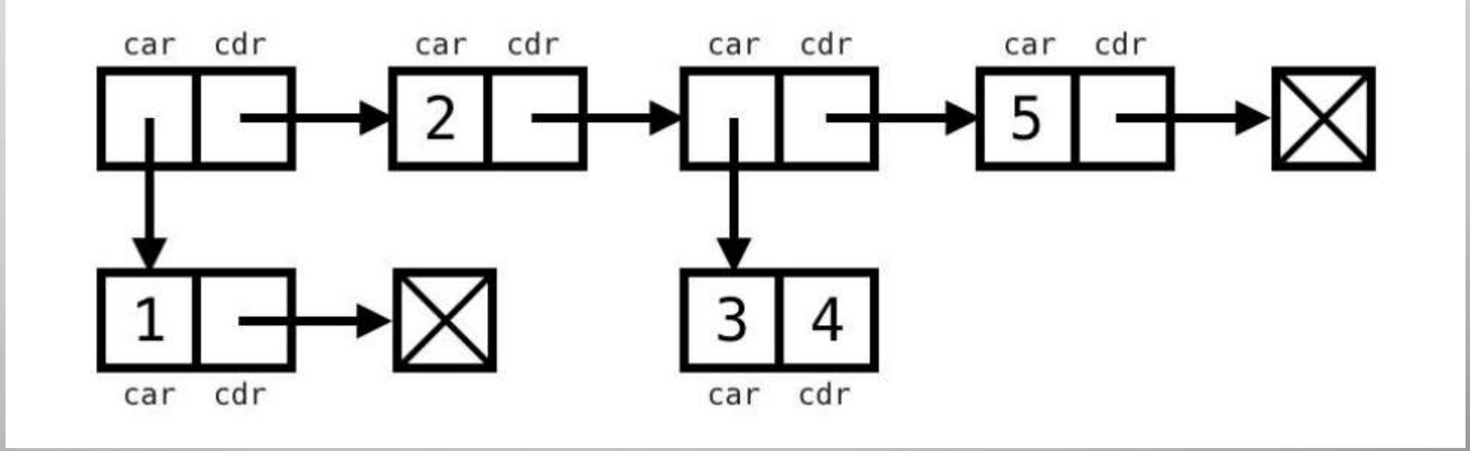
LISTS

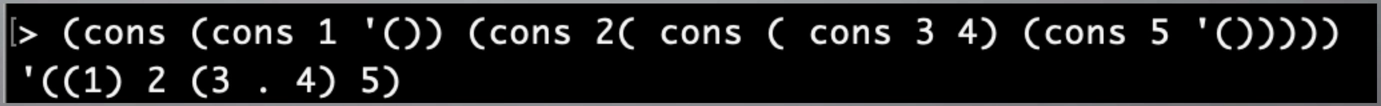
- Scheme uses linked lists, similar to OCaml and Prolog:



- To create a list, you can use (list 1 2 3) or ‘(1 2 3)
- To access the head, you can use (car my-list) or (first my-list)
- To access the tail, you can use (cdr my-list) or (rest my-list)
- Empty list: ‘()

EXERCISE

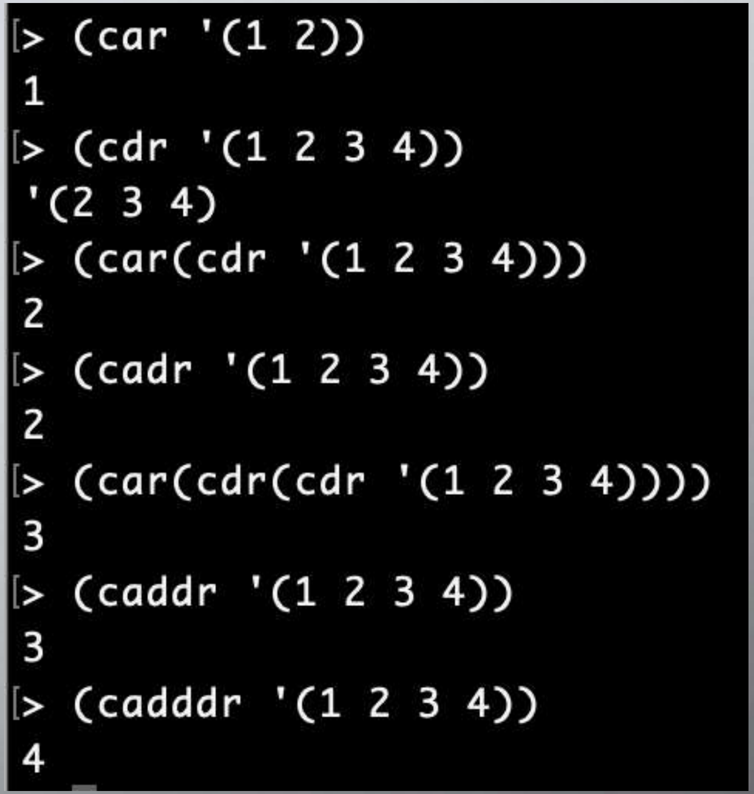




```
[> (cons (cons 1 '()) (cons 2( cons ( cons 3 4) (cons 5 '()))))
'((1) 2 (3 . 4) 5)
```

LISTS

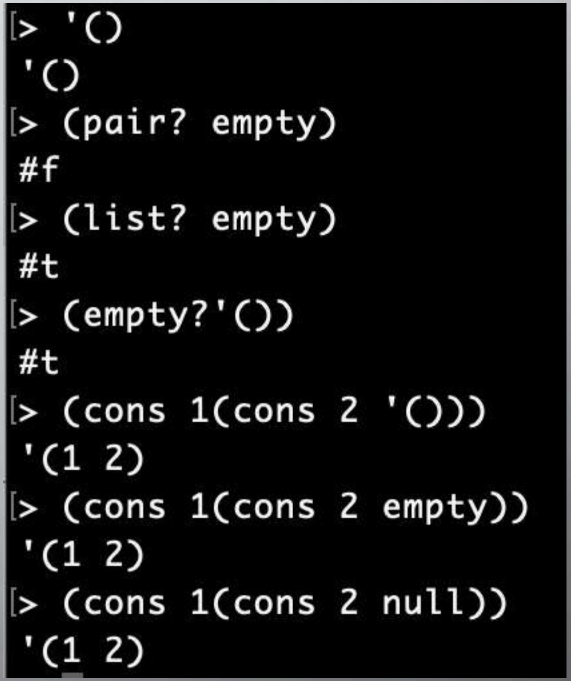
- car
  - Head of list
- cdr
  - Tail of list
- Cadr
  - “car of cdr”
  - 2nd element of list
- cadar
  - "car of the cdr of the car”



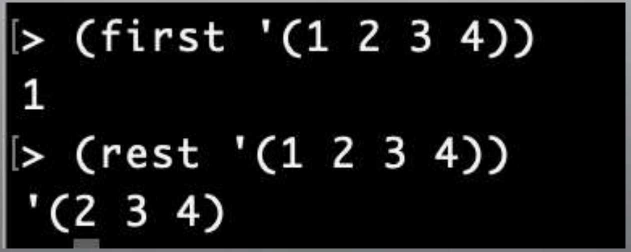
```
[> (car '(1 2))
1
[> (cdr '(1 2 3 4))
 '(2 3 4)
[> (car(cdr '(1 2 3 4)))
2
[> (cadr '(1 2 3 4))
2
[> (car(cdr(cdr '(1 2 3 4))))
3
[> (caddr '(1 2 3 4))
3
[> (cadddr '(1 2 3 4))
4
```

empty, first and rest

- empty will give an empty list '().
- Use empty? To make sure list is non-empty when using first and rest.
- First is same as car but only works on non-empty lists
- Rest is same as cdr but only works on non-empty lists



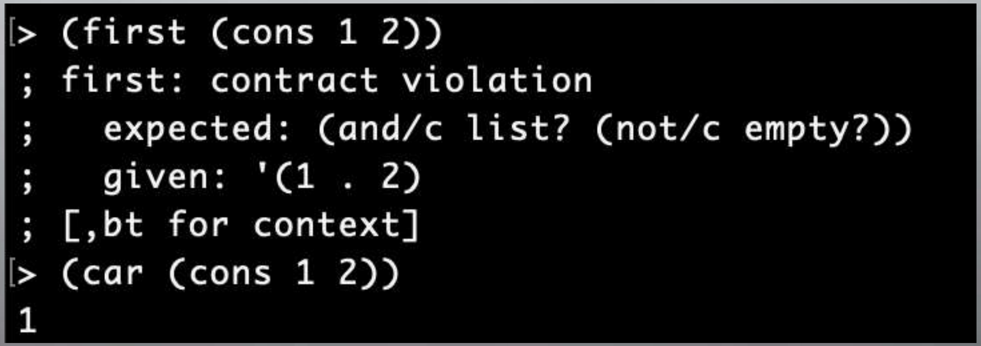
```
[> '()
'()
[> (pair? empty)
#f
[> (list? empty)
#t
[> (empty?'())
#t
[> (cons 1(cons 2 '()))
'(1 2)
[> (cons 1(cons 2 empty))
'(1 2)
[> (cons 1(cons 2 null))
'(1 2)
```



```
[> (first '(1 2 3 4))
1
[> (rest '(1 2 3 4))
'(2 3 4)
```

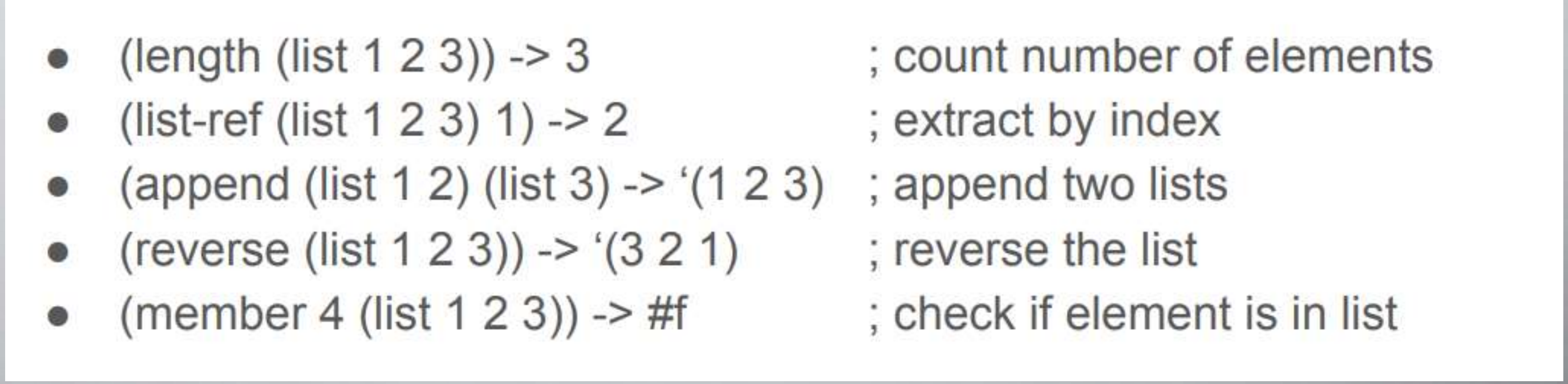
WHAT’S THE DIFFERENCE?

- In terms of what they do, **car** and **cdr** are equivalent to **first** and **rest.**
- Car and cdr work with pairs as well as lists.
- First and rest only work for lists.



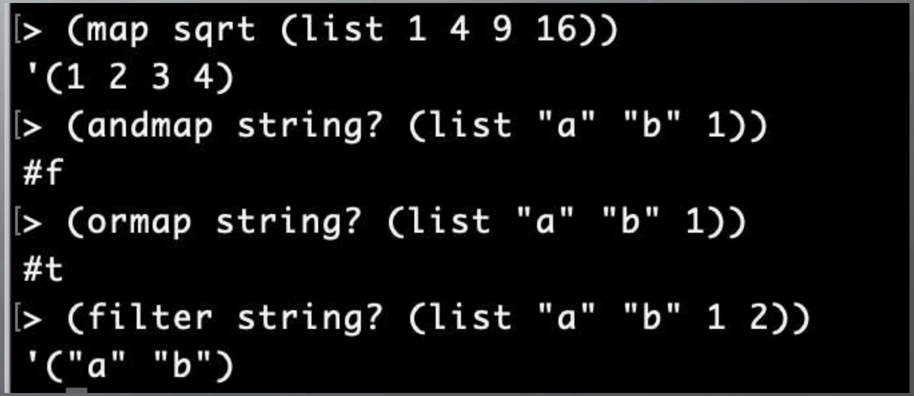
```
[> (first (cons 1 2))
; first: contract violation
;   expected: (and/c list? (not/c empty?))
;   given: '(1 . 2)
; [,bt for context]
[> (car (cons 1 2))
1
```

LIST OPERATIONS



- (length (list 1 2 3)) -> 3 ; count number of elements
- (list-ref (list 1 2 3) 1) -> 2 ; extract by index
- (append (list 1 2) (list 3) -> ‘(1 2 3) ; append two lists
- (reverse (list 1 2 3)) -> ‘(3 2 1) ; reverse the list
- (member 4 (list 1 2 3)) -> #f ; check if element is in list

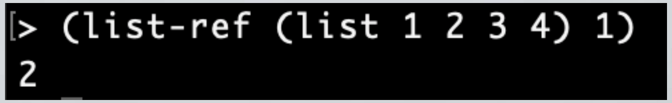
- There are predefined list loops as well:
  - map
  - filter
  - andmap
  - ormap



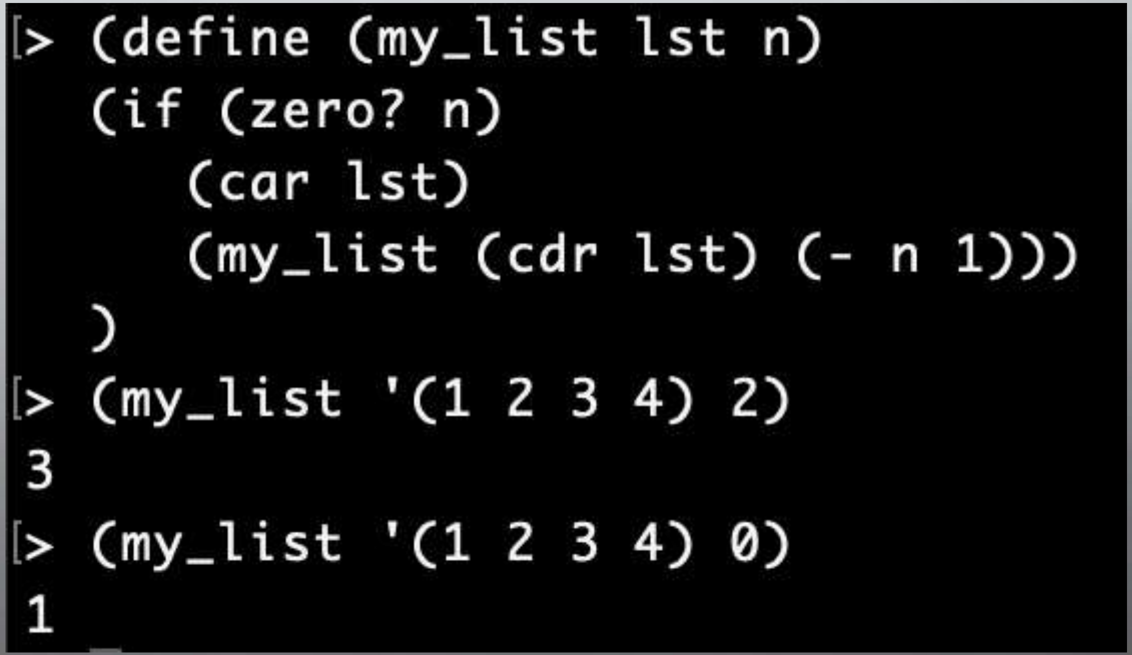
```
> (map sqrt (list 1 4 9 16))
'(1 2 3 4)
> (andmap string? (list "a" "b" 1))
#f
> (ormap string? (list "a" "b" 1))
#t
> (filter string? (list "a" "b" 1 2))
'("a" "b")
```

SCHEME
LISTS
EXAMPLE
EXERCISES

ITERATING A LIST (LOOPS)

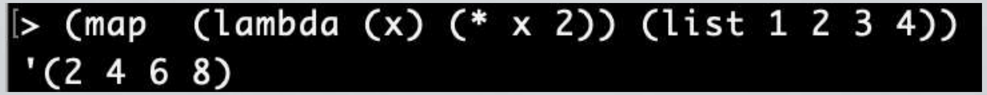


```
[> (list-ref (list 1 2 3 4) 1)
2
```



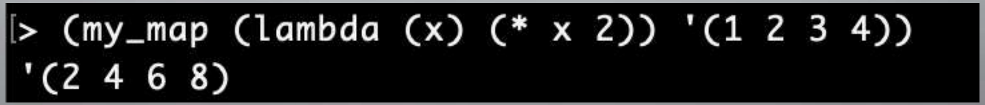
```
[> (define (my_list lst n)
   (if (zero? n)
      (car lst)
      (my_list (cdr lst) (- n 1)))
   )
[> (my_list '(1 2 3 4) 2)
3
[> (my_list '(1 2 3 4) 0)
1
```

LETS MAKE THE MAP FUNCTION



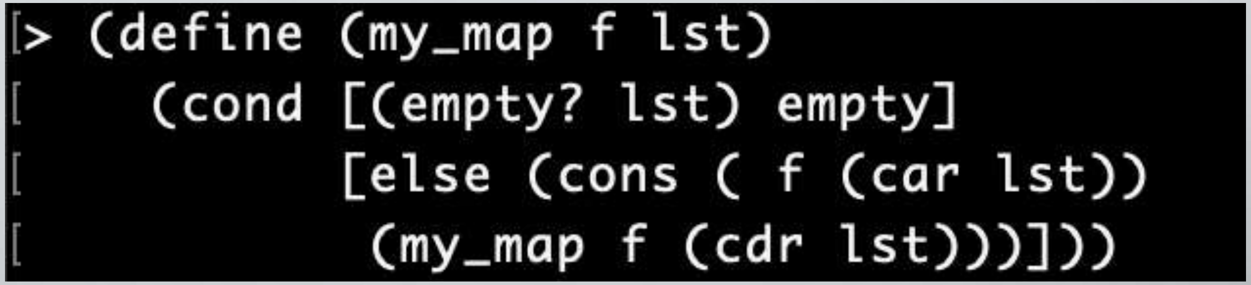
```
[> (map  (lambda (x) (* x 2)) (list 1 2 3 4))
'(2 4 6 8)
```

- Takes a function f, and a list, and returns a new list that has the results of applying f to each element in the list.

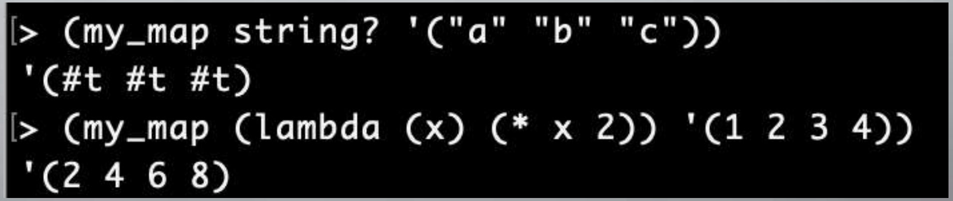


```
[> (my_map (lambda (x) (* x 2)) '(1 2 3 4))
'(2 4 6 8)
```

MAP FUNCTION: SOLUTION



```
[> (define (my_map f lst)
[    (cond [(empty? lst) empty]
[          [else (cons ( f (car lst))
[           (my_map f (cdr lst)))]))
```



```
[> (my_map string? '("a" "b" "c"))
'(#t #t #t)
[> (my_map (lambda (x) (* x 2)) '(1 2 3 4))
'(2 4 6 8)
```

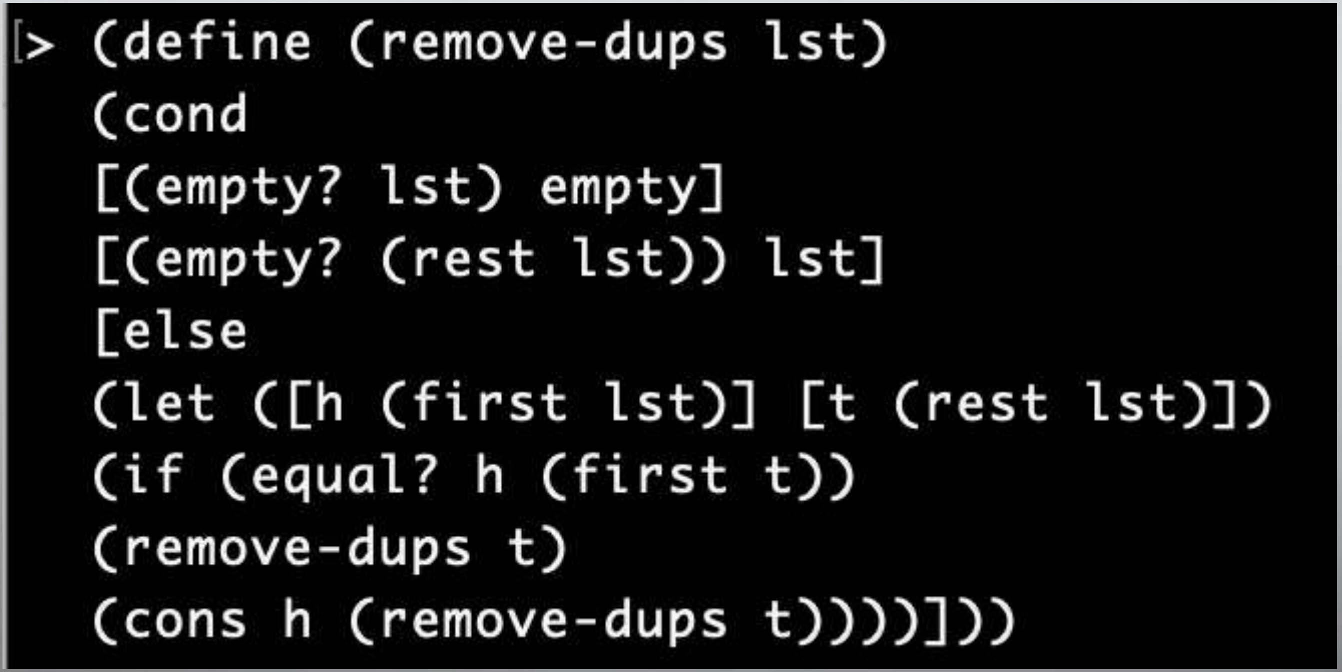
Is this solution tail recursive? If not, how could we make it?

REMOVING CONSECUTIVE DUPLICATES

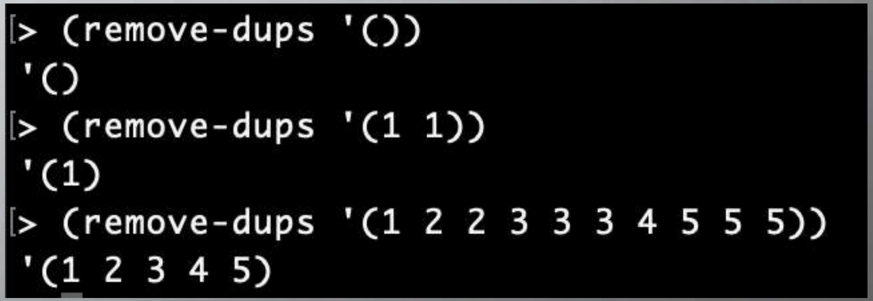
- Let’s remove the consecutive duplicates in a list:
- > (remove-dups (list "a" "b" "b" "b" "c" "c"))

'("a" "b" "c")

REMOVING CONSECUTIVE DUPLICATES: SOLUTION

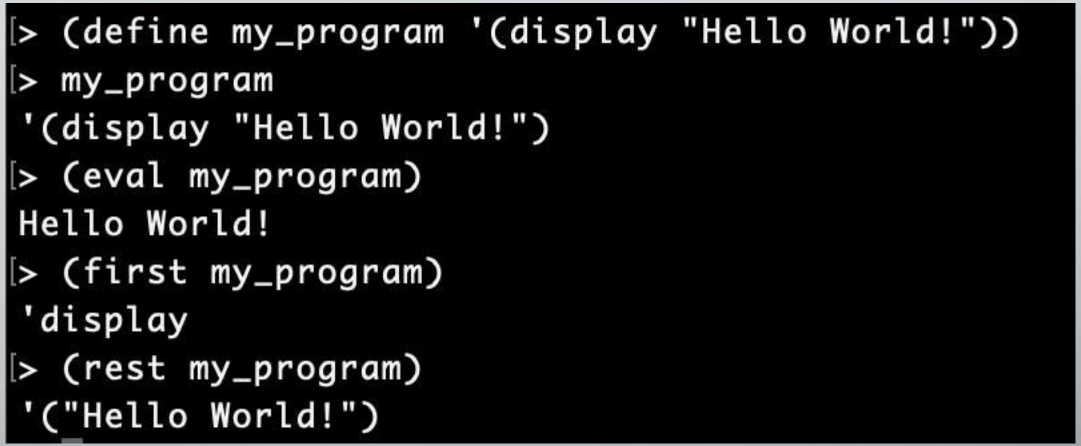


```
[> (define (remove-dups lst)
   (cond
   [(empty? lst) empty]
   [(empty? (rest lst)) lst]
   [else
   (let ([h (first lst)] [t (rest lst)])
   (if (equal? h (first t))
   (remove-dups t)
   (cons h (remove-dups t))))]))
```



```
[> (remove-dups '())
'()
[> (remove-dups '(1 1))
'(1)
[> (remove-dups '(1 2 2 3 3 3 4 5 5 5))
'(1 2 3 4 5)
```

PROGRAMS AS LISTS



```
[> (define my_program '(display "Hello World!"))
[> my_program
'(display "Hello World!")
[> (eval my_program)
Hello World!
[> (first my_program)
'display
[> (rest my_program)
'("Hello World!")
```

- Note that ‘(<list contents>) is a shorthand for (quote (<list contents>))

#HW5

- Task: Write a Scheme code difference analyzer
- Your function receives two Scheme expressions and returns an expression that combines similar parts of the expressions
- Can be used for Plagiarism Detection.

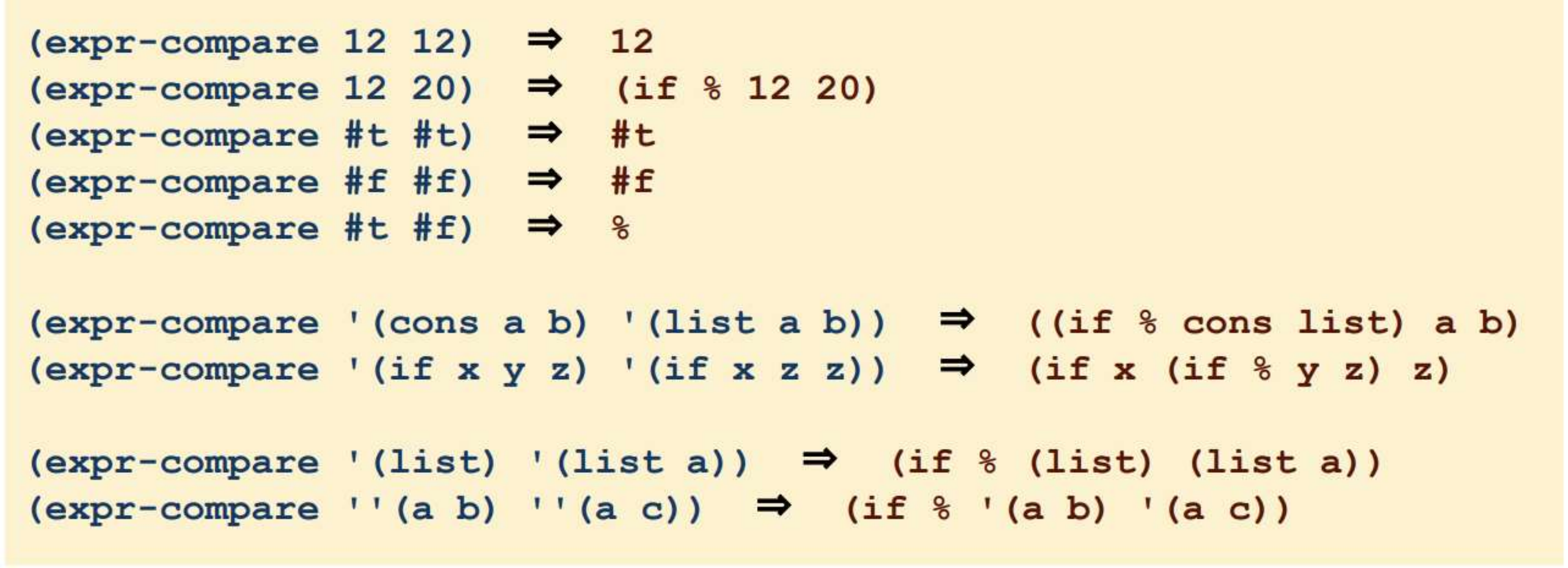
#HW5

- (expr-compare x y)
  - Check the structure of x and y
  - In places where different:
    - Replace with an if statement selecting the desired code to run (% variable determines which expression should be executed)
    - If the same, leave the same

#HW5

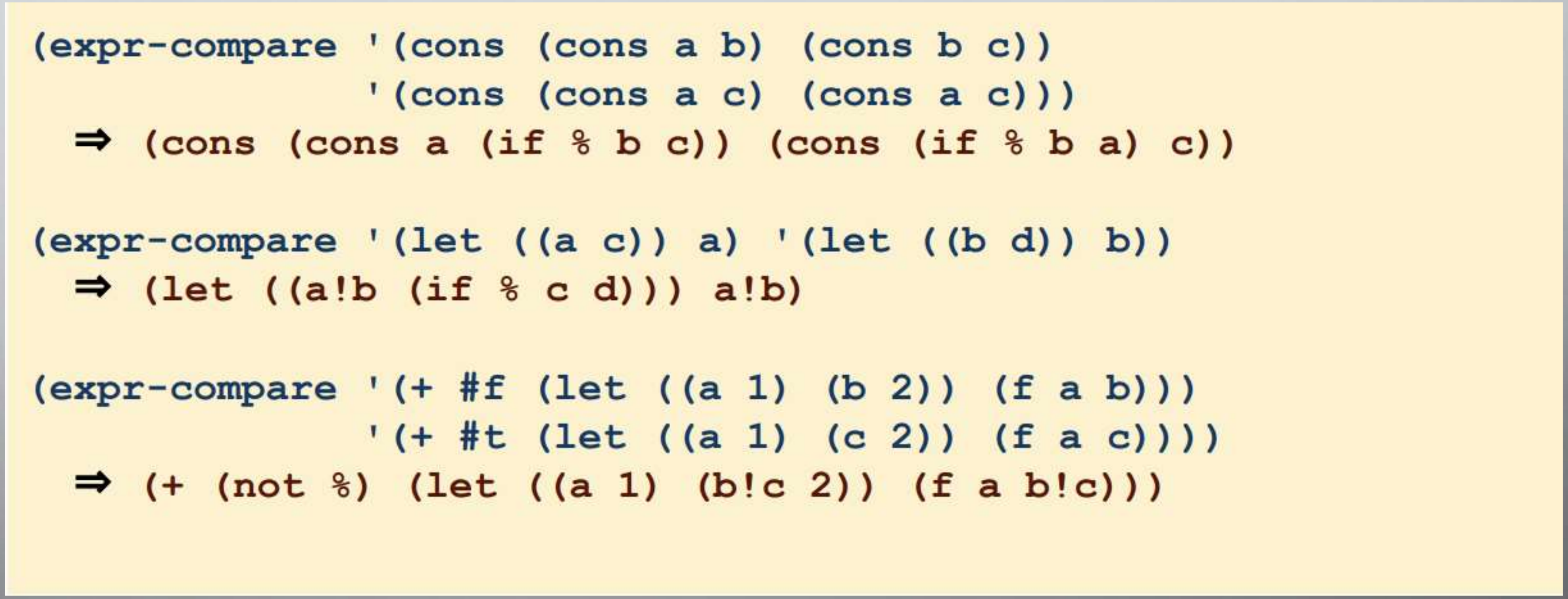
- (expr-compare x y)
- Check the structure of x and y
  - If same but different names for bound variables (declared in a let or lambda expression)
  - Replace each instance with combination of the two names separated by ‘!’ (ex: X!Y)

#HW5



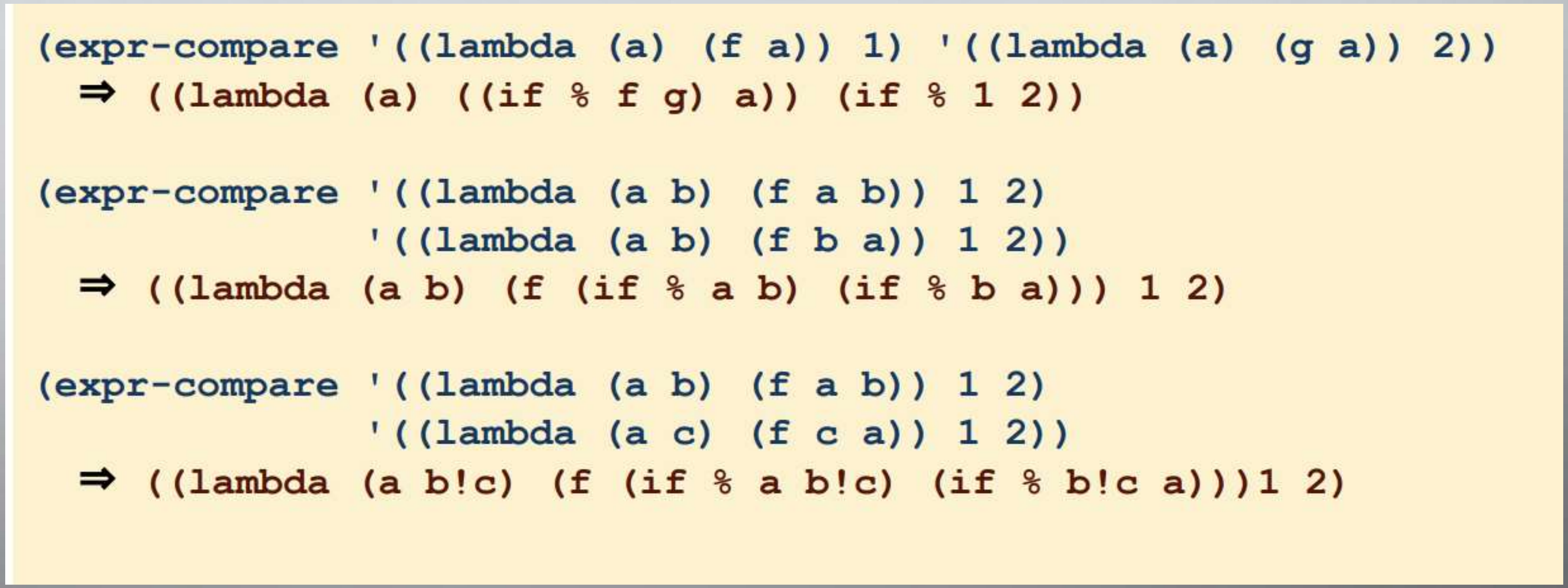
```
(expr-compare 12 12)  ⇒  12
(expr-compare 12 20)  ⇒  (if % 12 20)
(expr-compare #t #t)  ⇒  #t
(expr-compare #f #f)  ⇒  #f
(expr-compare #t #f)  ⇒  %

(expr-compare '(cons a b) '(list a b))  ⇒  ((if % cons list) a b)
(expr-compare '(if x y z) '(if x z z))  ⇒  (if x (if % y z) z)

(expr-compare '(list) '(list a))  ⇒  (if % (list) (list a))
(expr-compare ''(a b) ''(a c))  ⇒  (if % '(a b) '(a c))
```

#HW5



```
(expr-compare '(cons (cons a b) (cons b c))
              '(cons (cons a c) (cons a c)))
  ⇒ (cons (cons a (if % b c)) (cons (if % b a) c))

(expr-compare '(let ((a c)) a) '(let ((b d)) b))
  ⇒ (let ((a!b (if % c d))) a!b)

(expr-compare '(+ #f (let ((a 1) (b 2)) (f a b)))
              '(+ #t (let ((a 1) (c 2)) (f a c))))
  ⇒ (+ (not %) (let ((a 1) (b!c 2)) (f a b!c)))
```

#HW5



```
(expr-compare '((lambda (a) (f a)) 1) '((lambda (a) (g a)) 2))
  ⇒ ((lambda (a) ((if % f g) a)) (if % 1 2))

(expr-compare '((lambda (a b) (f a b)) 1 2)
              '((lambda (a b) (f b a)) 1 2))
  ⇒ ((lambda (a b) (f (if % a b) (if % b a))) 1 2)

(expr-compare '((lambda (a b) (f a b)) 1 2)
              '((lambda (a c) (f c a)) 1 2))
  ⇒ ((lambda (a b!c) (f (if % a b!c) (if % b!c a)))1 2)
```

#HW5

- Required to implement:
  - (expr-compare x y)
  - (test-expr-compare x y)
  - (test-expr-x) and (test-expr-y)

USEFUL LINKS:

- https://download.racket-lang.org/racket-v7.1.html
- https://docs.racket-lang.org/guide/index.html
- Useful Reading:
    - https://classes.soe.ucsc.edu/cmps112/Spring03/languages/scheme/SchemeTutorialA.html
    - https://docs.racket-lang.org/
    - https://docs.racket-lang.org/guide/eval.html
    - https://docs.racket-lang.org/racket-cheat/index.html
    - http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-12.html#node_sec_Temp_14
    - https://stackoverflow.com/questions/34984552/what-is-the-difference-between-quote-and-list