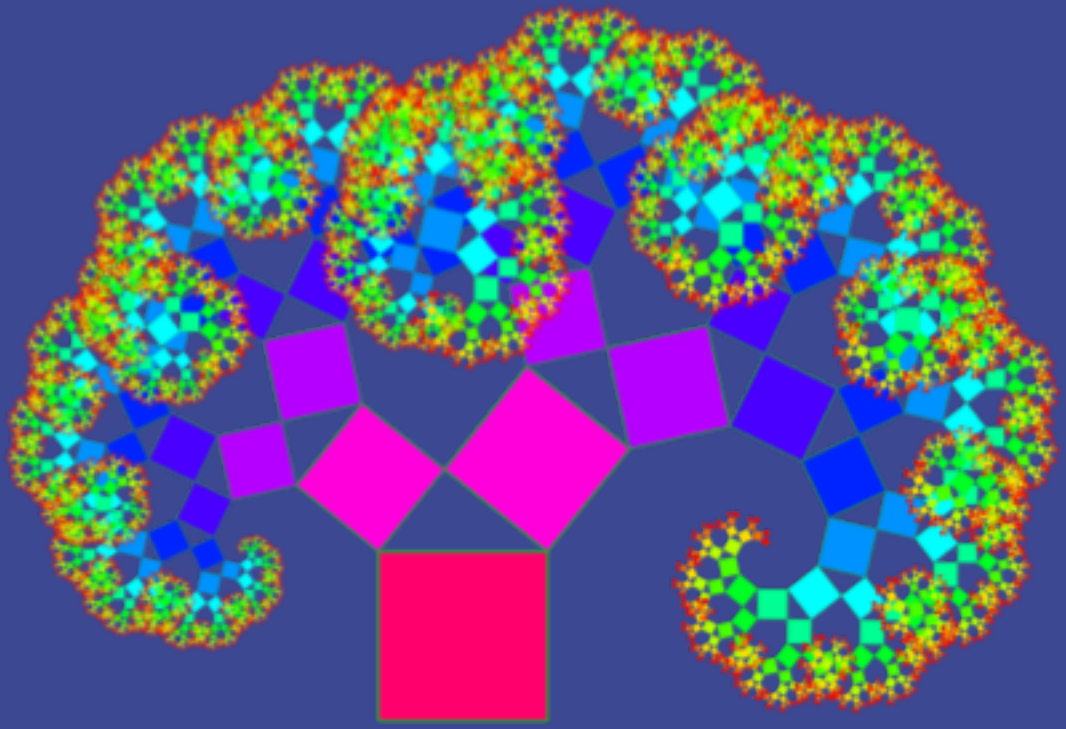
CS 152
Programming Paradigms

Scheme



Today

- More Function Definition Examples
- Recursion in Scheme
- Local bindings with *le*

Example: square

```
;;; Function square:  number -> number
;;; Returns the square of a given number
(define
  (square x)
  (* x x))
> (square 8)
64
> (square 2  3)
square: arity mismatch; the expected number of arguments does not
match the given number  expected: 1  given: 2
```

```
(define
 (<name> <formal parameters>)
  <body>)
```

Example: average

```
;;; Function average:  number number -> number
;;;  Returns the average of two numbers
(define
   (average first second)
   (/ (+ first second) 2))
```

```
(define
 (<name> <formal parameters>)
  <body>)
```

```
> (average 80 90)
85
```

Example: passing?

```
;;; Predicate passing?: number -> boolean
;;; Returns #t if the grade is >= 70
;;; Your function definition here

> (passing? 90)
#t
> (passing? 30)
#f
```

```
(define
 (<name> <formal parameters>)
  <body>)
```

Example: passing?

```
;;; Predicate passing?: number -> boolean
;;; Returns #t if the grade is >= 70
(define
  (passing? grade)
  (>= grade 70))
```

There is no need for the if special form.

```
> (passing? 90)
#t
> (passing? 30)
#f
```

Programming Techniques in Scheme

- Scheme relies on recursion to perform loops
- "cdr down and cons up": apply the operation recursively to the tail of a list and then use *cons* to construct a new list with the current result

Recursive Functions

- Base Case: problem size is as small as it can be

  numbers: 0 or 1?

  list: '() : null?

- Recursive Rule: formulate the problem in terms of one or more smaller problems.

  numbers: n - 1, n -2, etc…

  lists: car and cdr

Example: factorial

```
(factorial 5)
120
A. (define (factorial n) (* n (- n 1)))
B. (define (factorial n) (* n  (factorial (- n 1))
C. (define (factorial n)
       (if (= 0 n)
         1
         (* n (factorial (- n 1)))))
```

Example: member?

;;; Predicate member?: element list-> Boolean
;;; Returns #t if the list contains the element and #f otherwise

(member? 5 '(1 3 2 "hello")) -> #f
(member? 5 '(1 3 2 "hello" 5)) -> #t

- Base Case: problem size is as small as it can be?
- Recursive Rule: how do we compute (member? element xs) assuming that we know how to compute
  (member? element (cdr xs))

Example: member?

```
;;; Predicate member?: element list-> Boolean
;;; Returns #t if the list contains the element and #f otherwise
(define (member? x xs)
  (cond ((null? xs) #f)
        ((equal? x (car xs)) #t)
        (else (member? x (cdr xs)))))
```

= is for numbers only
we need to use equal?

Example: our-append

;;; Function our-append: list list -> list
;;; Returns list with all the elements of the two lists in order

(our-append '(1 2 3) '(4 5 6)) -> (1 2 3 4 5 6)

- Base Case: problem size is as small as it can be? Which list determines our problem size?
- Recursive Rule: Recursive Rule: how do we compute (our-append xs ys) assuming that we know how to compute
  (our-append (cdr xs) ys)

Example: our-append

```
;;; Function our-append: list list -> list
;;; Returns list with all the elements of the two lists in order
(define (our-append xs ys)
  (if (null? xs)
      ys
      (cons (car xs) (our-append (cdr xs) ys))))
```

count

;;; Function count: element list -> number
;;; Returns the count of the given element in the list

(count 5 '("hello" 5 6 9 5 2)) -> 2

(count 7 '("hello" 6 9 5 2)) -> 0

(count 'hello '()) -> 0

- Base Case: problem size is as small as it can be?
- Recursive Rule: how do we compute (count element xs) assuming that we know how to compute
  (count element (cdr xs))

Example: count

```
(count 5 '("hello" 5 6 9 5 2)) -> 2
(count 7 '("hello" 6 9 5 2)) -> 0

(define (count x xs)
  (cond ((null? xs) 0)
        ((equal? x (car xs)) (+ 1 (count x (cdr xs))))
        (else (count x (cdr xs)))))
```

Quote of the Day

"In order to understand recursion, one must first understand recursion."

What is the base case? 😀

Your turn

- 9 breakout rooms
- sum_of_squares: rooms 1, 2, 3
- all-positive: rooms 4, 5, 6
- remove: rooms 7, 8, 9

Your Turn: sum-of-squares

```
;;; Function sum-of-squares: list of numbers -> number
;;; Returns the sum of the squares of the elements in the list


(sum-of-squares '(0 1 2 3)) -> 14
(sum-of-squares '()) -> 0
```

Your Turn: all-positive?

```
;;; Predicate all-positive?: list of numbers -> boolean
;;; Returns #t if no element is less than 0 and #f otherwise
```

```
(allpositive? '(0 1 2 3)) -> #t
(allpositive? '(0 1 -2 3)) -> #f
(allpositive? '()) -> #t
```

Logical operator and:
(and expr1 expr2 …)

Your Turn: remove

;;; Function remove: element list -> list

;;; Removes all occurrences of the element from the given list

(remove 3 '(1 2 3 4 5 4 3 2 1)) -> (1 2 4 5 4 2 1)

(remove 9 '(1 2 3 4 5 4 3 2 1)) -> (1 2 3 4 5 4 3 2 1)

Solution: sum-of-squares

```
;;; Function sum-of-squares: list of numbers -> number

;;; Returns the sum of the squares of the elements in the list
(define (sum-of-squares xs)
  (if (null? xs)
      0
      (+ (* (car xs) (car xs)) (sum-of-squares (cdr xs)))))
```

Solution: all-positive?

```
;;; Predicate all-positive?: list of numbers -> boolean
;;; Returns #t if no element is less than 0 and #f otherwise
(define (all-positive? xs)
  (if (null? xs)
      #t
      (and (>= (car xs) 0 ) (all-positive? (cdr xs)))))
```

Solution: remove

```
;;; Function remove:  element list -> list
;;; Removes all occurrences of the element from the given list
(define (remove x xs)
  (cond ((null? xs)'())
        ((equal? (car xs) x) (remove x (cdr xs)))
        (else (cons (car xs)  (remove x (cdr xs))))))
```

Example: index

Return the index of the first occurrence of the given element in the list. If the element is not in the list, the function returns -1

```
(index 5 '("hello" 6 9 5 2 5))
3
(index 7 '("hello" 6 9 5 2))
-1
```

Example: index

```
;;; Function index:  element list -> number
;;; Returns the index of the first occurrence
;;; of the given element in the list.
;;; If the  element is not in the list, the function returns -1
(define (index x xs)
  (cond ((null? xs) -1)
        ((equal? x (car xs)) 0) ; we found our element
        ((equal? (index x (cdr xs)) -1) -1) ; element not in cdr
        (else (+ 1 (index x (cdr xs)))))) ; element is in the cdr
```

Example: index

```
;;; Function index: element list -> number
;;; Returns the index of the first occurrence
;;; of the given element in the list.
;;; If the element is not in the list, the function returns -1
(define (index x xs)
  (cond ((null? xs) -1)
        ((equal? x (car xs)) 0) ; we found our element
        ((equal? (index x (cdr xs)) -1) -1) ; element not in cdr
        (else (+ 1 (index x (cdr xs)))))) ; element is in the cdr
```

Block Scope with let…

The special form let introduces a list of local variables for use within its body:

```
(let
    (
      (variable1 value1)
      (variable2 value2)
       …
    )
    body)
```

Block Scope with let…

```
(let
  (
    (x 1)
    (y 2)
    (z 3)
  )
  (+ x y z))
```

iClicker:
What does the let expression evaluate to?

Block Scope with let…

```
(let ((x 1) (y 2))
   (let ((z x))
      (+ z y)))
```

iClicker:
What does the let expression evaluate to?

Block Scope with let…

```
(let  ((x 1)
       (y 2)
       (z x) )
        (+ z y)))
```

Error:
x: undefined;
cannot reference an identifier before its definition

Block Scope with let*…

```
(let*  ((x 1)
        (y 2)
        (z x) )
        (+ z y)))
```

In a let* expression, the bindings are performed sequentially. This let* expression evaluates to 3.

Syntactic Sugar

```
(let
  (
    (x 1)
    (y 2)
    (z 3)
  )
 (+ x y z))

((lambda (x y z) (+ x y z)) 1 2 3)
```

Example: index

```
;;; Returns the index of the first occurrence
;;; of the given element in the list.
;;; If the  element is not in the list, the function returns -1
(define (index x xs)
  (cond ((null? xs) -1)
        ((equal? x (car xs)) 0) ; we found our element
        (else
          (let ((index-rest (index x (cdr xs))))
            (if (= index-rest -1)
                -1
                (+ 1 index-rest))))))
```

To Do

- Homework 3
  - Team assignment
  - Due September 15