Programming Paradigms

Lecture 2. Functional programming in Racket

Test Nº1

See Moodle

Outline

- Recap: Simple definitions and expressions
- The Substitution Model
- Lists and recursion
- Tail recursion

Recap: definitions in Racket

```
(define a-disk
  (colorize (disk 90) "blue"))
```

Recap: definitions in Racket

special form

```
(define a-disk
  (colorize (disk 90) "blue"))
```

Recap: definitions in Racket

special form

identifier

```
(define a-disk
  (colorize (disk 90) "blue"))
```

Recap: definitions in Racket

special form

identifier

```
(define a-disk
  (colorize (disk 90) "blue"))
```

expression

Recap: function definitions in Racket

```
(define (square width color)
  (colorize
   (filled-rectangle width width)
   color))
```

Recap: function definitions in Racket

identifier

```
(define (square width color)
  (colorize
   (filled-rectangle width width)
   color))
```

Recap: function definitions in Racket

identifier

formal arguments

```
(define (square width color)
  (colorize
   (filled-rectangle width width)
   color))
```

Recap: function definitions in Racket

identifier

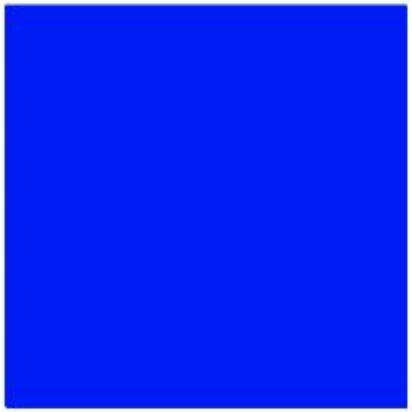
formal arguments

```
(define (square width color)
  (colorize
    (filled-rectangle width width)
    color))
```

body (expression)

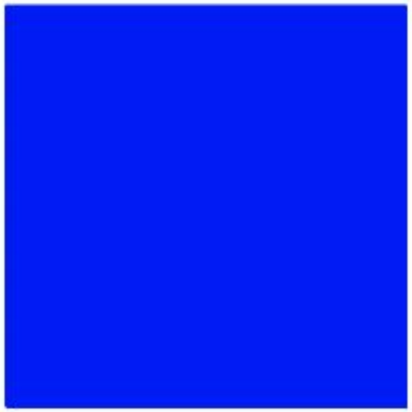
Recap: function calls in Racket

```
(square 100 "blue")

(square (+ 50 50) "blue")
```



Recap: function calls in Racket

```
(square 100 "blue")

(square (+ 50 50) "blue")
```



Recap: conditionals

```
(define (quantity-to-text word n)
  (cond
    [(= n 1)
     (string-append "one " word)]
    [(<= 2 n 3)
     (string-append
      "a couple of " word "s")]
    [else
     (string-append
      "many " word "s")]))
```

Recap: conditionals

```
(define (quantity-to-text word n)
  (cond
    [(= n 1)
     (string-append "one " word)]
    [(<= 2 n 3)
     (string-append
      "a couple of " word "s")]
    [else
     (string-append
      "many " word "s")]))
```

Recap: conditionals

```
(define (quantity-to-text word n)
  (cond
    [(= n 1)
     (string-append "one " word)]
    [(<= 2 n 3)
     (string-append
      "a couple of " word "s")]
    [else
     (string-append
      "many " word "s")]))
```

Recap: conditionals

```
(quantity-to-text "apple"  1)
; "one apple"

(quantity-to-text "orange" 2)
; "a couple of oranges"

(quantity-to-text "banana" 4)
; "many bananas"
```

Recap: anonymous functions

```
(define (twice f x)
  (f (f x)))
```

Recap: anonymous functions

```
(define (twice f x)
  (f (f x)))

(twice sqrt 16) ; 2
```

Recap: anonymous functions

```
(define (twice f x)
  (f (f x)))

(twice (lambda (x) (* x x)) 2) ; 16
```

Recap: anonymous functions

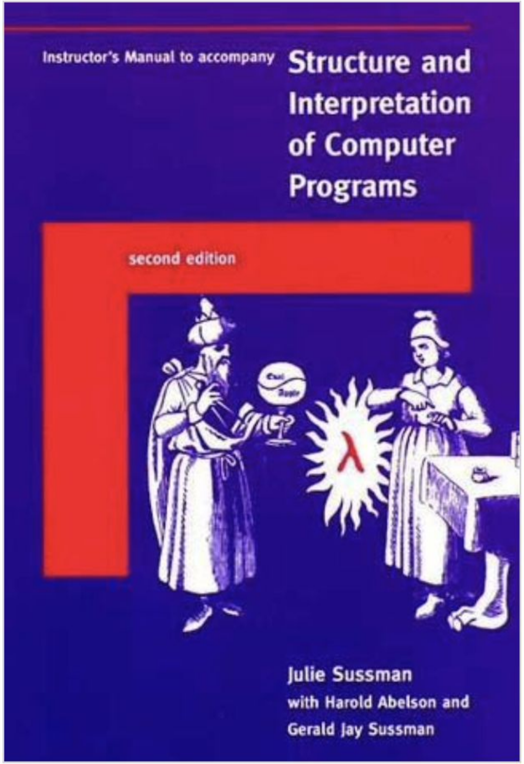
```
(define (twice f x)
  (f (f x)))

(twice (lambda (x) (* x x)) 2) ; 16
```

formal arguments

body (expression)

SICP, the wizard book

Structure and Interpretation of Computer Programs



https://mitpress.mit.edu/sites/default/files/sicp/index.html

The Substitution Model

```
(define (square x) (* x x))
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f z)
  (sum-of-squares (+ z 2) (* z 3)))
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f z)
  (sum-of-squares (+ z 2) (* z 3)))
```

```
(f 5)
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f z)
  (sum-of-squares (+ z 2) (* z 3)))
```

```
(f 5)
```

```
(sum-of-squares (+ z 2) (* z 3))
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f z)
  (sum-of-squares (+ z 2) (* z 3)))
```

```
(f 5)
```

```
(sum-of-squares (+ 5 2) (* 5 3))
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f z)
  (sum-of-squares (+ z 2) (* z 3)))
```

```
(sum-of-squares (+ 5 2) (* 5 3))
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f z)
  (sum-of-squares (+ z 2) (* z 3)))
```

```
(sum-of-squares 7 15)
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f z)
  (sum-of-squares (+ z 2) (* z 3)))
```

```
(sum-of-squares 7 15)
```

```
(+ (square x) (square y))
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f z)
  (sum-of-squares (+ z 2) (* z 3)))
```

```
(sum-of-squares 7 15)
```

```
(+ (square 7) (square 15))
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f z)
  (sum-of-squares (+ z 2) (* z 3)))
```

```
(+ (square 7) (square 15))
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f z)
  (sum-of-squares (+ z 2) (* z 3)))
```

```
(+ (* 7 7) (square 15))
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f z)
  (sum-of-squares (+ z 2) (* z 3)))
```

```
(+ 49 (square 15))
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f z)
  (sum-of-squares (+ z 2) (* z 3)))
```

```
(+ 49 (* 15 15))
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f z)
  (sum-of-squares (+ z 2) (* z 3)))
```

```
(+ 49 225)
```

The Substitution Model

```
(define (square x) (* x x))

(define (sum-of-squares x y)
  (+ (square x) (square y)))

(define (f z)
  (sum-of-squares (+ z 2) (* z 3)))
```

274

The Substitution Model: anonymous functions

```
(define (twice f x)
  (f (f x)))

(twice (lambda (x) (* x x)) 2)
```

The Substitution Model: anonymous functions

```
(define (twice f x)
  (f (f x)))

(twice (lambda (x) (* x x)) 2)
((lambda (x) (* x x)) ((lambda (x) (* x x)) 2))
```

The Substitution Model: anonymous functions

```
(define (twice f x)
  (f (f x)))

(twice (lambda (x) (* x x)) 2)
((lambda (x) (* x x)) ((lambda (x) (* x x)) 2))
((lambda (x) (* x x)) (* 2 2))
```

The Substitution Model: anonymous functions

```
(define (twice f x)
  (f (f x)))

(twice (lambda (x) (* x x)) 2)
((lambda (x) (* x x)) ((lambda (x) (* x x)) 2))
((lambda (x) (* x x)) (* 2 2))
((lambda (x) (* x x)) 4)
```

The Substitution Model: anonymous functions

```
(define (twice f x)
  (f (f x)))

(twice (lambda (x) (* x x)) 2)
((lambda (x) (* x x)) ((lambda (x) (* x x)) 2))
((lambda (x) (* x x)) (* 2 2))
((lambda (x) (* x x)) 4)
(* 4 4)
```

The Substitution Model: anonymous functions

```
(define (twice f x)
  (f (f x)))

(twice (lambda (x) (* x x)) 2)
((lambda (x) (* x x)) ((lambda (x) (* x x)) 2))
((lambda (x) (* x x)) (* 2 2))
((lambda (x) (* x x)) 4)
(* 4 4)
16
```

Lists

```
(list 1 2 3 4 5)

(list "apples" "bananas" "oranges")

(list 1 (list 2 3) (list (list)))
```

Function on lists

```
(define example
  (list "apples" "bananas" "oranges"))

(length example)
(list-ref example 1)
(reverse example)
(append example example example)
```

Deconstructing lists

```
(define example
  (list "apples" "bananas" "oranges"))

(first example) ; "apples"
(rest example)  ; '("bananas" "oranges")

(car example) ; "apples"
(cdr example) ; '("bananas" "oranges")
```

Constructing lists

```
empty                    ; '()
```

Constructing lists

```
empty                ; '()

(cons 1 (list 2 3)) ; '(1 2 3)
```

Constructing lists

```
empty                  ; '()

(cons 1 (list 2 3)) ; '(1 2 3)

(cons 1 (cons 2 (cons 3 empty)))
```

Checking structure of a list

```
(empty? (list 2 3))                ; #f

(empty? (rest (rest (list 2 3))))  ; #t

(cons? (list 2 3))                 ; #t
```

Checking structure of a list

```
(empty? (list 2 3))                ; #f

(empty? (rest (rest (list 2 3))))  ; #t

(cons? (list 2 3))                 ; #t
```

Recursive functions over lists

```
(define (my-sum lst)
   ...)
```

Recursive functions over lists

```
(define (my-sum lst)
  (cond
    [(empty? lst) ...]
    [else ...]))
```

Recursive functions over lists

```
(define (my-sum lst)
  (cond
    [(empty? lst) 0]
    [else ...]))
```

Recursive functions over lists

```
(define (my-sum lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
             (my-length (rest lst)))]))
```

Recursive functions over lists

```
(define (my-sum lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
             (my-length (rest lst)))]))

(my-sum (list 1 2 3)) ; 6
```

Tail recursion

```
(define (my-sum lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
                  (my-length (rest lst)))]))

(my-sum (list 1 2 3))
```

Tail recursion

```
(define (my-sum lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
                (my-length (rest lst)))]))
```

```
(my-sum (list 1 2 3))
(+ 1 (my-sum (list 2 3)))
```

Tail recursion

```
(define (my-sum lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
                (my-length (rest lst)))]))
```

```
(my-sum (list 1 2 3))
(+ 1 (my-sum (list 2 3)))
(+ 1 (+ 2 (my-sum (list 3))))
```

Tail recursion

```
(define (my-sum lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
             (my-length (rest lst)))]))
```

```
(my-sum (list 1 2 3))
(+ 1 (my-sum (list 2 3)))
(+ 1 (+ 2 (my-sum (list 3))))
(+ 1 (+ 2 (+ 3 (my-sum empty))))
```

Tail recursion

```
(define (my-sum lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
             (my-length (rest lst)))]))
```

```
(my-sum (list 1 2 3))
(+ 1 (my-sum (list 2 3)))
(+ 1 (+ 2 (my-sum (list 3))))
(+ 1 (+ 2 (+ 3 (my-sum empty))))
(+ 1 (+ 2 (+ 3 0)))
```

Tail recursion

```
(define (my-sum lst)
  (define (helper lst current)
    (cond
      [(empty? lst) current]
      [else (helper (rest lst)
                    (+ current (first lst)))]))
  (helper lst 0))
```

Tail recursion

```
(define (my-sum lst)
  (define (helper lst current)
    (cond
      [(empty? lst) current]
      [else (helper (rest lst)
                    (+ current (first lst)))]))
  (helper lst 0))

(my-sum (list 1 2 3))
```

Tail recursion

```
(define (my-sum lst)
  (define (helper lst current)
    (cond
      [(empty? lst) current]
      [else (helper (rest lst)
                    (+ current (first lst)))]))
  (helper lst 0))

(my-sum (list 1 2 3))
(helper (list 1 2 3) 0)
```

Tail recursion

```
(define (my-sum lst)
  (define (helper lst current)
    (cond
      [(empty? lst) current]
      [else (helper (rest lst)
                    (+ current (first lst)))]))
  (helper lst 0))

(my-sum (list 1 2 3))
(helper (list 1 2 3) 0)
(helper (list 2 3) (+ 0 1))
```

Tail recursion

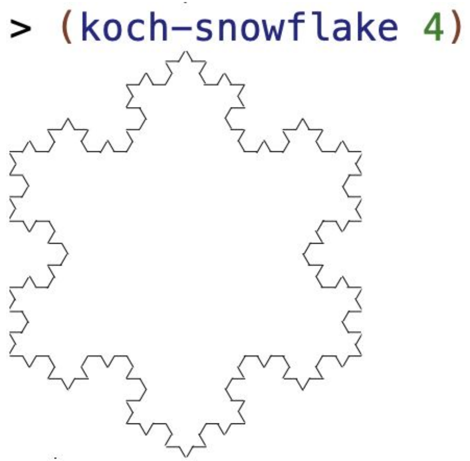
```
(define (my-sum lst)
  (define (helper lst current)
    (cond
      [(empty? lst) current]
      [else (helper (rest lst)
                    (+ current (first lst)))]))
  (helper lst 0))

(my-sum (list 1 2 3))
(helper (list 1 2 3) 0)
(helper (list 2 3) (+ 0 1))
(helper (list 2 3) 1)
...
```

Using apply

```
(define (my-sum lst)
  (apply + lst))



(my-sum (list 1 2 3 4))
```

Homework

1. Read **SICP 1.2 Procedures and the Processes They Generate**
2. Solve **exercises 1.11, 1.14, 1.16, 1.26** from **SICP**
3. Implement a function in Racket that renders a Koch snowflake of given rank:



References

1. The Substitution Model — SICP 1.1.5
2. Racket essentials 2.3–2.4

Mud cards