Lecture 2. Introduction to Racket.
Expressions, Pairs, and Lists. Substitution Model

Nikolai Kudasov
Programming Paradigms (Fall 2025)
Innopolis, Russia

Lab of Programming Languages and Compilers

INNOPOLIS
UNIVERSITY

Quiz for Lecture 1

Outline

1. Racket Basics: Simple Definitions and Expressions
2. The Substitution Model
3. Lists
4. Recursive Functions over Lists

Racket Basics: Simple Definitions and Expressions

About Racket

1. Racket is dynamically typed functional programming language
2. Racket is one of the popular dialects of LISP, more specifically, a dialect of Scheme
3. Racket has a dedicated IDE support (called DrRacket)
4. We will use Racket to learn about
    - programming with (pure) functions
    - managing local state with accumulator parameter
    - working with higher-order functions
    - iterators and list comprehensions
    - thinking about programs via the Substitution Model

Racket Definitions

special form

identifier

Racket
Lisp — (f x y z)

f(x, y, z)

f x y z

Read "Quick: An Introduction to Racket with Pictures"[1].

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

expression

[1]See https://docs.racket-lang.org/quick/index.html.

Racket Function Definitions

function name

formal argument

Read "Quick: An Introduction to Racket with Pictures"[2].

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

[2] See https://docs.racket-lang.org/quick/index.html.

Racket Function Calls

Read “Quick: An Introduction to Racket with Pictures”[3].

```
(square 100 "blue")
```

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

[3] See https://docs.racket-lang.org/quick/index.html.

Racket 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")]))
```

special form

[condition body]

(<= 2 n 3)

(+ 2 n 3)

$2 \leq n \leq 3$

$2 + n + 3$

Racket Conditionals (example calls)

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

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

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

Anonymous Functions

Read “Quick: An Introduction to Racket with Pictures”[4].

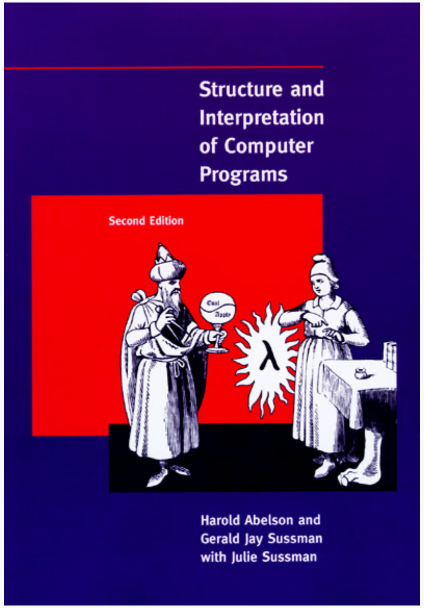
```
(define (twice f x)
  (f (f x)))
```

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

[4]See https://docs.racket-lang.org/quick/index.html.

The Substitution Model

The Wizard Book



Harold Abelson and Gerald Jay Sussman (1996). **Structure and interpretation of computer programs.** The MIT Press

The Substitution Model

The Substitution Model for Procedure Application

> *To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.*
>
> — SICP (Abelson and Sussman 1996, §1.1.5)

The Substitution Model for Procedure Application

> *To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.*
>
> — SICP (Abelson and Sussman 1996, §1.1.5)

1. The Substitution Model helps us think and **reason about programs** (semantically), **not** to understand how the interpreter works!

The Substitution Model

The Substitution Model for Procedure Application

> *To apply a compound procedure to arguments, evaluate the body of the procedure with each formal parameter replaced by the corresponding argument.*
>
> — SICP (Abelson and Sussman 1996, §1.1.5)

1. The Substitution Model helps us think and **reason about programs** (semantically), **not** to understand how the interpreter works!
2. We will crucially depend on *referential transparency*, without it the Substitution Model no longer works in general, see "The Cost of Introducing Assignment" (Abelson and Sussman 1996, §3.1.3).

The Substitution Model (example 1)

```
(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 (example 1)

```
(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 (example 1)

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

The Substitution Model (example 1)

```scheme
(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)))
```

```scheme
  (f 5)
= (sum-of-squares (+ 5 2) (* 5 3))
= (sum-of-squares 7 15)
= (+ (square 7) (square 15))
```

The Substitution Model (example 1)

```scheme
(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)))
```

```scheme
(f 5)
= (sum-of-squares (+ 5 2) (* 5 3))
= (sum-of-squares 7 15)
= (+ (square 7) (square 15))
= (+ (* 7 7) (* 15 15))
```

The Substitution Model (example 1)

```scheme
(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))
= (sum-of-squares 7 15)
= (+ (square 7) (square 15))
= (+ (* 7 7) (* 15 15))
= (+ 49 225)
```

The Substitution Model (example 1)

```scheme
(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)))
```

```scheme
(f 5)
= (sum-of-squares (+ 5 2) (* 5 3))
= (sum-of-squares 7 15)
= (+ (square 7) (square 15))
= (+ (* 7 7) (* 15 15))
= (+ 49 225)
= 274
```

The Substitution Model (example 2)

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

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

The Substitution Model (example 2)

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

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

= ((lambda (x) (* x x)) ((lambda (x) (* x x)) 2))
```

f   f

The Substitution Model (example 2)

```
(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 (example 2)

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

```scheme
(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 (example 2)

```
(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 (example 2)

```
(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
```

The Substitution Model (example 3a)

```
(define (make-twice f)
  (lambda (x) (f (f x))))
```

```
((make-twice g) z)
```

The Substitution Model (example 3a)

```
(define (make-twice f)
  (lambda (x) (f (f x))))
```

```
((make-twice g) z)
= ((lambda (x) (g (g x))) z)
```

The Substitution Model (example 3a)

```
(define (make-twice f)
  (lambda (x) (f (f x))))
```

```
((make-twice g) z)
= ((lambda (x) (g (g x))) z)
= (g (g z))
```

The Substitution Model (example 3a)

```
(define (make-twice f)
  (lambda (x) (f (f x))))
```

```
((make-twice g) z)
= ((lambda (x) (g (g x))) z)
= (g (g z))
```

Note that this works for any (pure) `g` and `z`.

The Substitution Model (example 3b)

```
(define (make-twice f)
  (lambda (x) (f (f x))))
```

```
(((make-twice make-twice) f) x)
```

The Substitution Model (example 3b)

```
(define (make-twice f)
  (lambda (x) (f (f x))))
```

```
(((make-twice make-twice) f) x)
= ((make-twice (make-twice f)) x)
```

((make-twice g) z)

→ (g (g z))

The Substitution Model (example 3b)

```
(define (make-twice f)
  (lambda (x) (f (f x))))
```

```
(((make-twice make-twice) f) x)
= ((make-twice (make-twice f)) x)
= ((make-twice f) ((make-twice f) x))
```

((make-twice g) z)

→ (g (g z))

The Substitution Model (example 3b)

```
(define (make-twice f)
  (lambda (x) (f (f x))))
```

```
(((make-twice make-twice) f) x)
= ((make-twice (make-twice f)) x)
= ((make-twice f) ((make-twice f) x))
= ((make-twice f) (f (f x)))
```

The Substitution Model (example 3b)

```scheme
(define (make-twice f)
  (lambda (x) (f (f x))))
```

```scheme
(((make-twice make-twice) f) x)
= ((make-twice (make-twice f)) x)
= ((make-twice f) ((make-twice f) x))
= ((make-twice f) (f (f x)))
= (f (f (f (f x))))
```

Lists

Racket Lists

Lists in Racket can be constructed using function `list`:

```
(list 1 2 3 4 5)

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

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

Lists in Racket are heterogeneous (may contain elements of different types).

Standard Functions on Lists

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

(length example)        ; 3
(list-ref example 1)    ; "bananas"
(reverse example)       ; '("oranges" "bananas" "apples")
(append example example)
; '("apples" "bananas" "oranges" "apples" "bananas" "oranges")
```

Deconstructing Lists

- `first` (or `car`) takes the first element of the list.
- `rest` (or `cdr`) takes the remaining part of the list.

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

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

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

Constructing Lists

`cons` constructs a new list given its first element (head) and its tail.

```
empty                    ; '()

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

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

Checking Structure of Lists

- `empty?` checks if a list is empty.
- `cons?` checks that a list is non-empty[5].

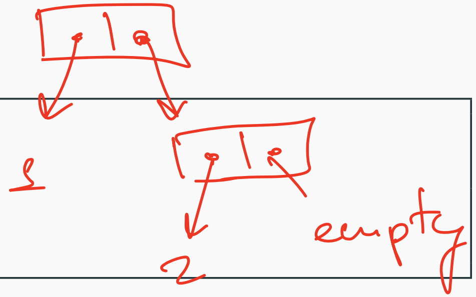
```
(empty? (list 2 3))                   ; #f

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

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

[5] Actually, this is a more general predicate, as we will see in a minute

Pairs in Racket

cons takes two values and constructs a pair:

```
(cons 1 2)              ; '(1 . 2)
(cons "hi" "world")     ; '("hi" . "world")
```



pair? is the same as cons?:

```
(pair? (cons 1 2))   ; #t
```

A list[6] either empty or a pair of a value and a list:

```
(cons 1 empty)          ; '(1)       same as '(1 . ())
(cons 1 (list 2 3))     ; '(1 2 3)   same as '(1 . (2 . (3)))
```

[6] See Racket Essentials §2.4.

List Components

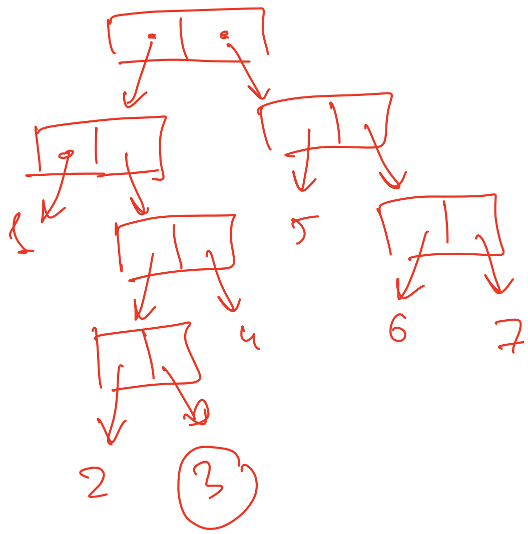
Helper functions exist[7] to extract a certain element of a list. There are also short-named combinations of `car` and `cdr`.

```
(first  (list 1 2 3)) ; 1
(second (list 1 2 3)) ; 2
(third  (list 1 2 3)) ; 3


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

```
(car    (list 1 2 3))
(cadr   (list 1 2 3))
(caddr  (list 1 2 3))

 cdr
(rest (list 1 2 3))
```

[7] See Racket Essentials §2.4.

car cdr



cadadr

Quotation

Note that Racket outputs many values with a single quote:

```
(list (list 1) (list 2 3) (list 4)) ; '((1) (2 3) (4))
```

This is shorthand for application of `quote`[8]:

```
(quote ((1) (2 3) (4)))             ; '((1) (2 3) (4))
'((1) (2 3) (4))                    ; '((1) (2 3) (4))
```

The argument of `quote` is **not** evaluated and is effectively treated as data (not code).

Note, that `quote` is **not** a function, it is a *special form* (a macro), and has to be explicitly applied.

[8] See Racket Essentials §2.4.

5 min break

Recursive Functions over Lists

Sum of a List (naïve recursive)

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

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

Sum of a List (naïve recursive)

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

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

Sum of a List (naïve recursive)

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

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

Sum of a List (naïve recursive)

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

Sum of a List (naïve recursive)

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

Sum of a List (naïve recursive)

```
(define (my-sum lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
             (my-sum (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)))
= (+ 1 (+ 2 3))
```

Sum of a List (naïve recursive)

```
(define (my-sum lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
             (my-sum (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)))
= (+ 1 (+ 2 3))
= (+ 1 5)
```

Sum of a List (naïve recursive)

```
(define (my-sum lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
             (my-sum (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)))
= (+ 1 (+ 2 3))
= (+ 1 5)
= 6
```

Sum of a List (naïve recursive)

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

- Two branches — one for empty list (base case), one for non-empty list (recursion).
- Simple definition, easy to verify (e.g. via Substitution Model).
- Recursive call is used as an argument to `+` (so `my-sum` is not tail recursive).
- This leads to a problem: a large expression is accumulated in memory before any addition can be computed.

Sum of a List (tail recursive with accumulator parameter)

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

Idea (accumulator parameter pattern):

- Introduce a tail recursive helper function with an additional argument `current` that will represent local state (intermediate sum value).
- Update `current` when doing a recursive call.
- Call the helper from the main function with some initial state.

Sum of a List (tail recursive with accumulator parameter)

```
(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))
```

Sum of a List (tail recursive with accumulator parameter)

```
(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)
```

Sum of a List (tail recursive with accumulator parameter)

```
(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)
```

Sum of a List (tail recursive with accumulator parameter)

```
(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)
= (helper (list 3) (+ 1 2)) = (helper (list 3) 3)
```

Sum of a List (tail recursive with accumulator parameter)

```
(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)
= (helper (list 3) (+ 1 2)) = (helper (list 3) 3)
= (helper (list) (+ 3 3)) = (helper (list) 6)
```

Sum of a List (tail recursive with accumulator parameter)

```
(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)
= (helper (list 3) (+ 1 2)) = (helper (list 3) 3)
= (helper (list) (+ 3 3)) = (helper (list) 6)
= 6
```

Accumulator Parameter Pattern vs Imperative Loop

Compare our Racket implementation against a version in Python using a `while`-loop:

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

```python
# Python (while-loop)
def my_sum(lst):
  current = 0
  while True:
    if len(lst) == 0:
      return current
    else:
      current = current + lst[0]
      lst = lst[1:]
```

Sum of a List (via higher-order functions)

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

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

Sum of a List (via higher-order functions)

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

```
(my-sum (list 1 2 3 4))
= (apply + (list 1 2 3 4))
```

Sum of a List (via higher-order functions)

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

```
(my-sum (list 1 2 3 4))
= (apply + (list 1 2 3 4))
= (+ 1 2 3 4)
```

Sum of a List (via higher-order functions)

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

```
(my-sum (list 1 2 3 4))
= (apply + (list 1 2 3 4))
= (+ 1 2 3 4)
= 10
```

Quick recap

1. **Substitution Model** helps us , but does not
2. **Substitution Model** in Racket (LISP) implements order of evaluation
3. **Tail recursion** helps with of recursive functions
4. **Accumulator parameter** is a

Quick recap

1. **Substitution Model** helps us think about function application, but does not

2. **Substitution Model** in Racket (LISP) implements order of evaluation
3. **Tail recursion** helps with of recursive functions
4. **Accumulator parameter** is a

Quick recap

1. **Substitution Model** helps us think about function application, but does not provide a description of how the interpreter works
2. **Substitution Model** in Racket (LISP) implements order of evaluation
3. **Tail recursion** helps with of recursive functions
4. **Accumulator parameter** is a

Quick recap

1. **Substitution Model** helps us think about function application, but does not provide a description of how the interpreter works
2. **Substitution Model** in Racket (LISP) implements applicative order of evaluation
3. **Tail recursion** helps with of recursive functions
4. **Accumulator parameter** is a

Quick recap

1. **Substitution Model** helps us think about function application, but does not provide a description of how the interpreter works
2. **Substitution Model** in Racket (LISP) implements applicative order of evaluation
3. **Tail recursion** helps with efficiency of recursive functions
4. **Accumulator parameter** is a

Quick recap

1. **Substitution Model** helps us think about function application, but does not provide a description of how the interpreter works
2. **Substitution Model** in Racket (LISP) implements applicative order of evaluation
3. **Tail recursion** helps with efficiency of recursive functions
4. **Accumulator parameter** is a pattern for handling local state in a purely functional manner

Reflection

Before we conclude, take a moment to reflect on Lecture 2, and complete the form at https://forms.gle/GFYNBaVVuCE619ue6.

What's Next

Today, in the labs you will

1. practice implementing explicitly recursive functions (naïve and with tail recursion)

Next week, we will delve into higher-order functions. To prepare:

1. Read SICP §1.2 *Procedures and the Processes They Generate* (Abelson and Sussman 1996, §1.2)
2. Solve exercises 1.11, 1.14, 1.16, 1.26 from SICP
3. Try implementing a Racket function that renders Koch snowflake[9]

[9] See https://en.wikipedia.org/wiki/Koch_snowflake.

Thank you!

References i

Abelson, Harold and Gerald Jay Sussman (1996). **Structure and interpretation of computer programs.** The MIT Press.