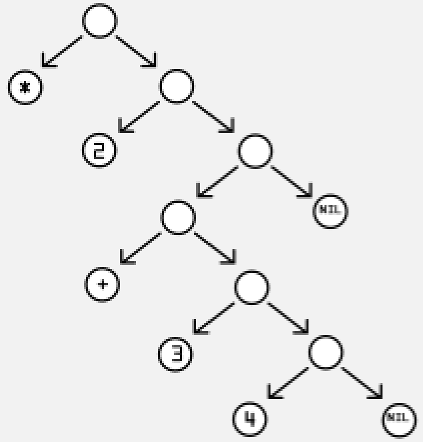
UMass Boston Computer Science

CS450 High Level Languages

Intertwined Data

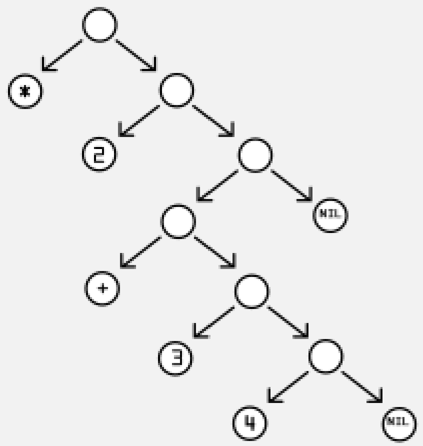
Thursday, March 26, 2026



lecture14

Logistics

- HW 7 out
  - due: Tues 3/31, 11am EST
  - "intertwined" data (today!)



S-expression (from `wikipedia`)

Previously

Randomness

[bracketed args] = optional

```
(random k [rand-gen]) → exact-nonnegative-integer?
  k : (integer-in 1 4294967087)
  rand-gen : pseudo-random-generator?
           = (current-pseudo-random-generator)
```

Optional args need Default value

Interlude: Optional and Keyword Arguments

- **Optional** (by position)

Optional arg

Default value

```
(define/contract (greet first [last "Chang"])
  (->* (string?) (string?) string?)
  (string-append "Hi " first " " last))
```

nonoptional arg(s)

Contract with optional arg

`(greet "Prof")` ⇨ `"Hi Prof Chang"`

`(greet "450" "student")` ⇨ `"Hi 450 student"`

- **Keyword** (by name)

Keyword arg

Default value

```
(define/contract (greet2 first #:last [last "Chang"])
  (->* (string?) (#:last string?) string?)
  (string-append "Hi " first " " last))
```

Contract with optional keyword arg

`(greet2 "Prof")` ⇨ `"Hi Prof Chang"`

`(greet2 "450" #:last "student")` ⇨ `"Hi 450 student"`

`(greet2 #:last "student" "450")` ⇨ `"Hi 450 student"`

Order of keyword arg(s) don't matter

Last Time

More Recursive Data Definitions: Trees

```
;; A Tree<X> is one of:
;; - empty
;; - (mk-node Tree<X> X Tree<X>)
(struct node [left data right])
;; Represents: a binary tree data structure
```

Last Time

Tree Template

```
;; A Tree<X> is one of:
;; - empty
;; - (mk-node Tree<X> X Tree<X>)
(struct node [left data right])
;; Represents: a binary tree data structure
```

```
;; tree-fn : Tree<X> -> ???
(define (tree-fn t)
  (cond
   [(empty? t) ...]
   [(node? t) ... (tree-fn (node-left t)) ...
                  ... (node-data t) ...
              ... (tree-fn (node-right t)) ...]))
```

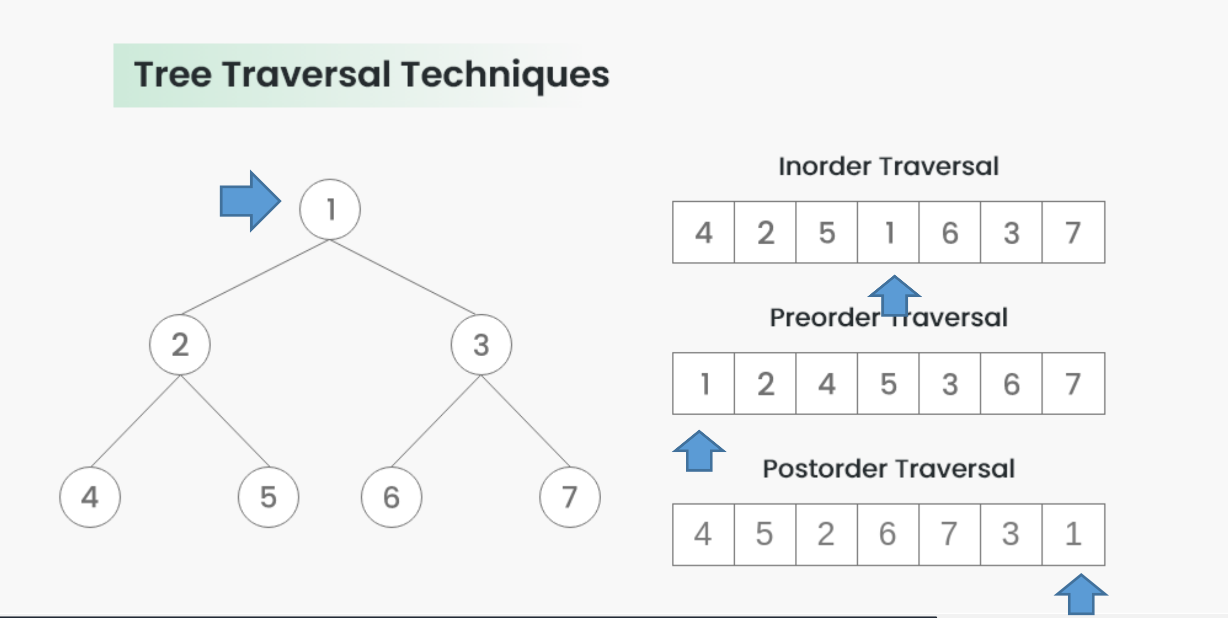
Template:
Recursive call(s) match recursion in data definition

Template:
cond clause for each itemization item

Template:
Extract pieces of compound data

Last Time

Tree Algorithms




```
;; tree->lst/in : Tree<X> -> List<X>
;; converts given tree to a list of values, by inorder
```

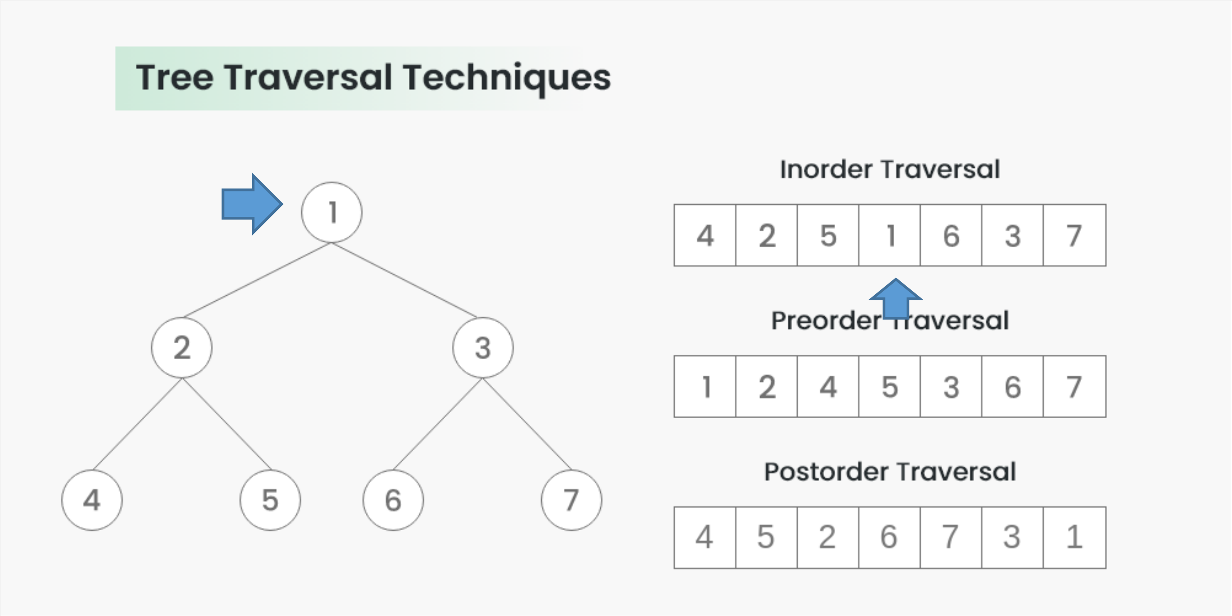
```
;; tree->lst/pre : Tree<X> -> List<X>
;; converts given tree to a list of values, by preorder
```

```
;; tree->lst/post : Tree<X> -> List<X>
;; converts given tree to a list of values, by postorder
```

Main difference: when to process root node

Last Time

In-order Traversal



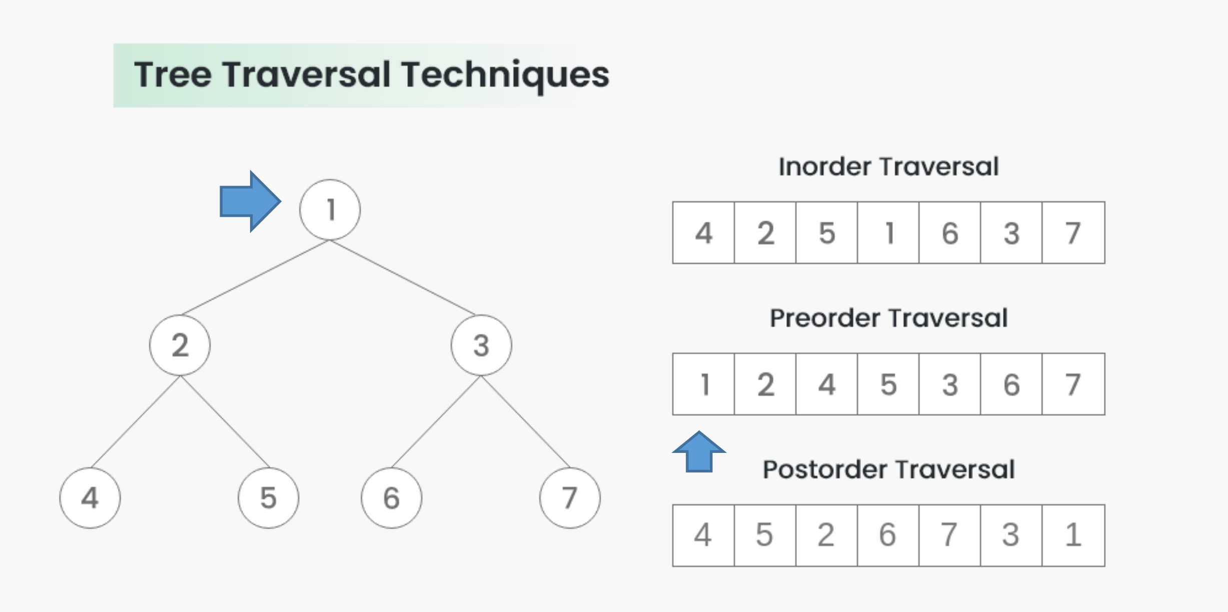

```
;; tree->lst/in : Tree<X> -> List<X>
;; converts given tree to a list of values, by inorder
```

```
(define (tree->lst/in t)
  (cond
    [(empty? t) empty]
    [(node? t) (append (tree->lst/in (node-left t))
                       (cons (node-data t)
                             (tree->lst/in (node-right t))))]))
```

To **fill in template** ...
figure out how to
"combine pieces"

Last Time

Pre-order Traversal



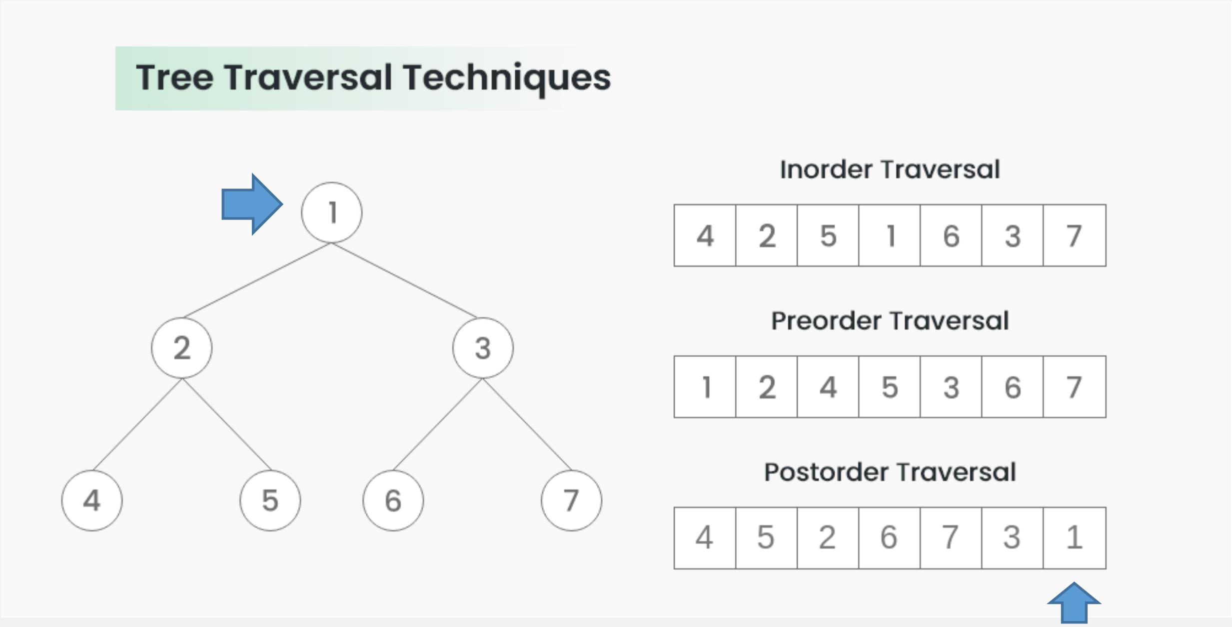

```
;; tree->lst/pre : Tree<X> -> List<X>
;; converts given tree to a list of values, by preorder
```

```
(define (tree->lst/pre t)
  (cond
    [(empty? t) empty]
    [(node? t) (cons (node-data t)
                 (append (tree->lst/pre (node-left t))
                         (tree->lst/pre (node-right t))))]))
```

To **fill in template** …
figure out how to
“**combine pieces**”

Last Time

Post-order Traversal




```
;; tree->lst/post : Tree<X> -> List<X>
;; converts given tree to a list of values, by postorder
```

```
(define (tree->lst/post t)
  (cond
    [(empty? t) empty]
    [(node? t) (append (tree->lst/post (node-left t))
                       (tree->lst/post (node-right t))
                       (list (node-data t)))]))
```

To **fill in template** ...
figure out how to
"combine pieces"

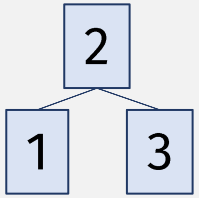
Last Time

Sometimes called **andmap** (for Racket lists) or **every** (for JS Arrays)

tree-all?

```
;; tree-all? : (X -> Boolean) Tree<X> -> Boolean
;; Returns true if given predicate returns true
;; for all values in given tree
```

```
(define TREE1 (node empty 1 empty))
(define TREE3 (node empty 3 empty))
(define TREE123 (node TREE1 2 TREE3))
```




```
(check-true (tree-all? (lambda (x) (< x 4)) TREE123))
```

Last Time

tree-all?

```
;; tree-all? : (X -> Boolean) Tree<X> -> Boolean
;; Returns true if given predicate returns true
;; for all values in given tree
```

```
(define (tree-all? p? t)
  (cond
    [(empty? t) true]
    [(node? t)
     (and (p? (node-data t))
          (tree-all? p? (node-left t))
          (tree-all? p? (node-right t)))]))
```

Template:
cond clause for each itemization item

Last Time

tree-all?

```
;; tree-all? : (X -> Boolean) Tree<X> -> Boolean
;; Returns true if given predicate returns true
;; for all values in given tree
```

```
(define (tree-all? p? t)
  (cond
    [(empty? t) true]
    [(node? t)
     (and (p? (node-data t))
          (tree-all? p? (node-left t))
          (tree-all? p? (node-right t)))]))
```

Last Time

tree-all?

```
;; tree-all? : (X -> Boolean) Tree<X> -> Boolean
;; Returns true if given predicate returns true
;; for all values in given tree
```

```
(define (tree-all? p? t)
  (cond
    [(empty? t) true]
    [(node? t)
     (and (p? (node-data t))
          (tree-all? p? (node-left t))
          (tree-all? p? (node-right t)))]))
```

Template:
Recursive call(s) match recursion in data definition

Template:
Extract pieces of compound data

Last Time

tree-all?

```
;; tree-all? : (X -> Boolean) Tree<X> -> Boolean
;; Returns true if given predicate returns true
;; for all values in given tree
```

```
(define (tree-all? p? t)
  (cond
    [(empty? t) true]
    [(node? t)
     (and (p? (node-data t))
          (tree-all? p? (node-left t))
          (tree-all? p? (node-right t)))]))
```

Combine the pieces with arithmetic to complete the function!

cond that evaluates to a boolean constant is just boolean arithmetic!

```
(define (tree-all? p? t)
  (or (empty? t)
      (and (p? (node-data t))
           (tree-all? p? (node-left t))
           (tree-all? p? (node-right t)))))
```

Tree Find?

- Do we have to search the entire tree?

Data Definitions With Invariants

```
;; A Tree<X> is one of:
;; - empty
;; - (mk-node Tree<X> X Tree<X>)
(struct node [left data right])
;; Represents: a binary tree data structure
```

(deep) predicate?

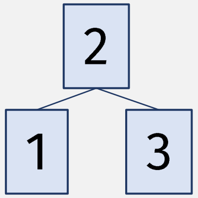
```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
```

;; Invariant 1: ∀ x ∈ left tree, x < node-data
;; Invariant 2: ∀ y ∈ right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST

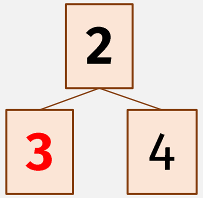
Valid BSTs

```
;; valid-bst? : Tree<X> -> Bool
;; Returns true if the tree is a BST
```

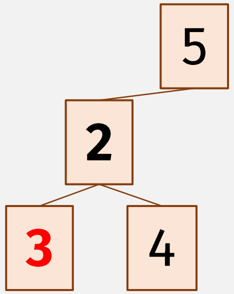
Valid



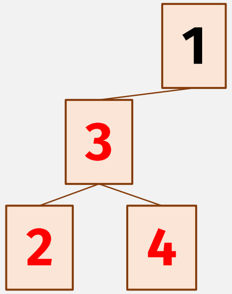
Not Valid



left value > root ☒



left values less than root ☑,
but left subtree not BST ☒



Left subtree is valid BST ☑,
but left values not less than root ☒

Valid BSTs

```
;; valid-bst? : Tree<X> -> Bool
;; Returns true if the tree is a BST
```

```
(define (valid-bst? t)
  (cond
    [(empty? t) true]
    [(node? t)
     (and (tree-all? (curry > (node-data t)) (node-left t))
          (tree-all? (curry <= (node-data t)) (node-right t))
          (valid-bst? (node-left t))
          (valid-bst? (node-right t)))]))
```

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀ x ∈ left tree, x < node-data
;; Invariant 2: ∀ y ∈ right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

cond that evaluates to a **boolean constant** is just **boolean arithmetic!**

```
(define (valid-bst? t)
  (or (empty? t)
      (and (tree-all? (curry > (node-data t)) (node-left t))
           (tree-all? (curry <= (node-data t)) (node-right t))
           (valid-bst? (node-left t))
           (valid-bst? (node-right t)))))
```

BUT … **requires multiple passes?**

One-pass `valid-bst?`

```
;; valid-bst/one-pass? : Tree<X> -> Bool
;; Returns true if the tree is a BST
```

```
(define (valid-bst/one-pass? t)
  (or (empty? t)
      (and (valid-bst/one-pass? (node-left t))
           (valid-bst/one-pass? (node-right t)))))
```

Where is `(node-data t)`??

One-pass **valid-bst?**

```
;; valid-bst/one-pass? : ??? Tree<X> -> Bool
;; Returns true if the tree is a BST
```

```
(define (valid-bst/one-pass? ??? t)
  (or (empty? t)
      (and (valid-bst/one-pass? ??? ??? (node-left t))
           (valid-bst/one-pass? ??? ??? (node-right t)))))
```

- **Need extra argument(s) …**
- … to **keep track** of **the** valid interval for each **node-data** value

One-pass **valid-bst?** - high-level style!

```
;; valid-bst/one-pass? : Tree<X> (X -> Bool) -> Bool
;; Returns true if (valid? (node-data t)) and subtrees are BSTs
```

```
(define (valid-bst/one-pass? valid? t)
  (or (empty? t)
      (and (valid? (node-data t))
           (valid-bst/one-pass? ???

                                 (node-left t))
           (valid-bst/one-pass? ???

                                 (node-right
```

valid? checks <u>valid interval</u> for **node-data** value

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀ x ∈ left tree, x < node-data
;; Invariant 2: ∀ y ∈ right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

One-pass **valid-bst?** - high-level style!

```
;; valid-bst/one-pass? : Tree<X> (X -> Bool) -> Bool
;; Returns true if (valid? (node-data t)) and subtrees are
```

```
(define (valid-bst/one-pass? valid? t)
  (or (empty? t)
      (and (valid? (node-data t))
           (valid-bst/one-pass?

                                   (curry > (node-data t))))
                        (node-left t))
           (valid-bst/one-pass? ???



                        (node-right
```

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀ x ∈ left tree, x < node-data
;; Invariant 2: ∀ y ∈ right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

Interlude

Currying

- A **curried** function is **partially applied to some** (not all) **args**
- **Result** is **another function**

NOTE: First argument is <u>first arg</u> to fn

```
(curry < 4)
;; = a function that returns true when given a number greater than 4
```

```
(lambda (x) (< 4 x))
```

```
(define (smaller-than lst thresh)
  (filter (lambda (x) (< x thresh)) lst))
```

```
(define (smaller-than lst thresh)
  (filter (curry > thresh) lst))
```

```
(define (smaller-than lst thresh)
  (filter (curryr < thresh) lst))
```

NOTE: First argument is <u>last arg</u> to fn

One-pass **valid-bst?** - high-level style!

```
;; valid-bst/one-pass? : Tree<X> (X -> Bool) -> Bool
;; Returns true if (valid? (node-data t)) and subtrees are
```

```
(define (valid-bst/one-pass? valid? t)
  (or (empty? t)
      (and (valid? (node-data t))
           (valid-bst/one-pass?

                                             (curry > (node-data t))))
                          (node-left t))
           (valid-bst/one-pass? ???


                          (node-right
```

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀ x ∈ left tree, x < node-data
;; Invariant 2: ∀ y ∈ right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

One-pass **valid-bst?** - high-level style!

```
;; valid-bst/one-pass? : Tree<X> (X -> Bool) -> Bool
;; Returns true if (valid? (node-data t)) and subtrees are
```

```
(define (valid-bst/one-pass? valid? t)
  (or (empty? t)
      (and (valid? (node-data t))
           (valid-bst/one-pass? (lambda (x)
                                  (and (valid? x)
                                       ((curry > (node-data t)) x)))
                                (node-left t))
           (valid-bst/one-pass? ???



                                (node-right
```

new "valid?"

Need to still check previous valid?

```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀ x ∈ left tree, x < node-data
;; Invariant 2: ∀ y ∈ right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

One-pass `valid-bst?` - high-level style!

```
;; valid-bst/one-pass? : Tree<X> (X -> Bool) -> Bool
;; Returns true if (valid? (node-data t)) and subtrees are
```

```
(define (valid-bst/one-pass? valid? t)
  (or (empty? t)
      (and (valid? (node-data t))
           (valid-bst/one-pass? (lambda (x)
                                  (and (valid? x)
                                       ((curry > (node-data t)) x))
                                (node-left t))
           (valid-bst/one-pass? (lambda (x)
                                  (and (valid? x)
                                       ((curry <= (node-data t)) x))
                                (node-right t)))))
```

new "valid?"

Need to still check previous `valid?`

```
(conjoin p1? p2?)
==
(λ (x) (and (p1? x) (p2? x)))
```

"conjoin" is function arithmetic that combines predicates

One-pass **valid-bst?** - high-level style!

```
;; valid-bst/one-pass? : Tree<X> (X -> Bool) -> Bool
;; Returns true if (valid? (node-data t)) and subtrees are
```

```
(define (valid-bst/one-pass? valid? t)
  (or (empty? t)
      (and (valid? (node-data t))
           (valid-bst/one-pass? (conjoin
                                    valid?
                                    (curry > (node-data t))    )
                                (node-left t))
           (valid-bst/one-pass? (conjoin
                                    valid?
                                    (curry <= (node-data t))    )
                                (node-right t)))))
```

```
(conjoin p1? p2?)
==
(λ (x) (and (p1? x) (p2? x)))
```

One-pass `valid-bst?`

```
;; valid-bst/one-pass? : ??? Tree<X> -> Bool
;; Returns true if the tree is a BST
```

```
(define (valid-bst/one-pass? ??? t)
  (or (empty? t)
      (and (valid-bst/one-pass? ??? ??? (node-left t))
           (valid-bst/one-pass? ??? ??? (node-right t)))))
```

- Need extra argument(s) …
- … to keep track of allowed `node-data` values

More generally:

- Tree traversal processes each node independently …
- Extra argument allows "remembering" information from other nodes

One-pass **valid-bst?** - high-level style!

```
;; valid-bst/p? : Tree<X> (X -> Bool) -> Bool
;; Returns true if (p? (node-data t)) = true, and t is a BST
```

```
(define (valid-bst/p? p? t)
  (or (empty? t)
      (and (p? (node-data t))
           (valid-bst/p? (conjoin p? (curry > (node-data t)))
                         (node-left t))
           (valid-bst/p? (conjoin p? (curry <= (node-data t)))
                         (node-right t)))))
```

Extra argument, to "remember" information (valid **node-data** values) from other nodes

"Extra argument" is an **accumulator** !

Previously

Design Recipe For Accumulator Functions

When a function needs **“extra information”:**

1. *Specify* **accumulator:**
   - Name
   - Signature
   - Invariant
2. *Define* internal “helper” fn with extra **accumulator** arg

(Helper fn does <u>not</u> need extra description, statement, or examples, if they are the same …)

3. *Call* “helper” fn , with <u>initial</u> accumulator value, from original fn

Valid BSTs – with accumulators!

Function needs "extra information" …

```
;; valid-bst? : Tree<X> -> Bool
;; Returns true if t is a BST
(define (valid-bst? t)

  ;; accumulator p? : (X -> Bool)
  ;; invariant: if t = (node l data r), p? checks valid range
  ;; for node-data, so (p? (node-data t)) is always true
  (define (valid-bst/p? p? t)
    (or (empty? t)
        (and (p? (node-data t))
             (valid-bst/p? (conjoin p? (curry > (node-data t)))
                           (node-left t))
             (valid-bst/p? (conjoin p? (curry <= (node-data t)))
                           (node-right t)))))

  (valid-bst/p? (lambda (x) true) t))
```

1. *Specify* **accumulator:** name, signature, invariant

2. *Define* internal "helper" fn with **accumulator** arg

3. *Call* "helper" fn, with initial **accumulator**

A Tree Data Example

- A very common **kind of data** that is a **Tree** is … **Programs**!
- Come up with **a Data Definition** for **…**
- **… valid Racket Programs**

Valid Racket Programs

Examples

- 1
- “one”
- (+ 1 2)

Data Def

```
;; A RacketProg is a:
;; - Number
;; - String
;; - ???
```

Valid Racket Programs

- 1
- “one”
- (+ 1 2)

```
;; A RacketProg is a:
;; - Atom
```

```
;; An Atom is a:
;; - Number
;; - String
```

```
;; - ???
```

Valid Racket Programs

- `(+ 1 2)`

List of … atoms?

“symbol”

```
;; A RacketProg is a:
;; - Atom
;; - List<Atom> ???
```

```
;; An Atom is a:
;; - Number
;; - String
;; - Symbol
```

Written with a single quote, e.g., ‘+

Valid Racket Programs

- ```
  (* (+ 1 2)
     (- 4 3))
  ```
  Tree?
- ```
  (* (+ 1 2)
     (- 4 3)
     (/ 10 5))
  ```

Each tree “node” is a list, of ... RacketProgs ??

But: how many values does each node have?? Unknown!

```
;; A RacketProg is a:
;; - Atom
;; - List<???>
```

```
;; - Tree<???>
```

```
;; An Atom is a:
;; - Number
;; - String
;; - Symbol
```

Valid Racket Programs

- `(* (+ 1 2)`
  `(- 4 3))`

Tree?

- `(* (+ 1 2)`
  `(- 4 3)`
  `(/ 10 5))`

Each tree "node" is a list, of ... RacketProgs ??

But: how many values does each node have??

```
;; A RacketProg is a:
;; - Atom
;; - ProgTree
```

```
;; An Atom is a:
;; - Number
;; - String
;; - Symbol
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
```

Recursive Data Def!

Valid Racket Programs

Also, **Intertwined** Data Defs!

```
;; A RacketProg is a:
;; - Atom
;; - ProgTree
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
```

Intertwined Data

- A <u>set</u> of Data Definitions that <u>reference each other</u>
- <u>Templates</u> should be **defined together** …

```
;; A RacketProg is a:
;; - Atom
;; - ProgTree
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
```

Intertwined Data

- A <u>set</u> of Data Definitions that <u>reference each other</u>
- <u>Templates</u> should be **defined together …**
  - … and should **reference each other's templates** (when needed)

```
;; A RacketProg is one of:
;; - Atom
;; - ProgTree
(define (prog-fn p) ...)
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
(define (ptree-fn t) ...)
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
(define (atom-fn a) ...)
```

???

Intertwined Templates

```
;; A RacketProg is one of:
;; - Atom
;; - ProgTree
(define (prog-fn s)
  (cond
   [(atom? s) ... (atom-fn s) ...]
   [else  ... (ptree-fn s) ...]))
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
(define (atom-fn a)
 (cond
   [(number? a) ... ]
   [(string? a) ... ]
   [(symbol? a) ... ]))
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons RacketProg ProgTree)
(define (ptree-fn t)
  (cond
   [(empty? t) ...]
   [else  ... (prog-fn (first t)) ... (ptree-fn (rest t)) ...]))
```

Intertwined data have intertwined templates!

A “Racket Prog” = S-expression!

```
;; A ~~RacketProg~~ Sexpr is one of:
;; - Atom
;; - ProgTree
```

```
(define (sexpr-fn s)
  (cond
   [(atom? s) ... (atom-fn s) ...]
   [else  ... (ptree-fn s) ...]))
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons ~~RacketProg~~ Sexpr ProgTree)
```

```
(define (ptree-fn t)
  (cond
   [(empty? t) ...]
   [else  ... (sexpr-fn (first t)) ... (ptree-fn (rest t)) ...]))
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
```

```
(define (atom-fn a)
 (cond
   [(number? a) ... ]
   [(string? a) ... ]
   [(symbol? a) ... ]))
```

S-expressions

- A common real-world data definition!
  - For representing code
  - Or any tree-like data / document

- Equivalent: XML

  Uses:
  - web API queries, e.g., RSS, Atom, Google, MS
  - Documents: MS Office documents, SVG images
  - Code: JSX (React)

- Similar: JSON

  Uses:
  - web API queries: Twitter, Facebook, Github
  - Documents: config files (yaml, node.js)
  - Code: JS objects!

- Submit: GradeScope

In-class Coding 3/26: Counting Symbols

```
;; A Sexpr is one of:
;; - Atom
;; - ProgTree
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
```

```
;; count : Symbol Sexpr -> Nat
;; Computes the number of times the given
;; symbol appears in the given s-expression
```

```
;; A ProgTree is one of:
;; - empty
;; - (cons Sexpr ProgTree)
```

```
;; count-atom : Symbol Atom -> Nat
;; ???
```

```
;; count-ptree : Symbol ProgTree -> Nat
;; ???
```

Counting Symbols

```
;; count : Symbol Sexpr -> Nat
;; Computes the number of times the given
;; symbol appears in the given s-expression
(define (count sym se)
  (cond
   [(atom? s) ... (atom-fn s) ...]
   [else  ... (ptree-fn s) ...]))
```

```
;; count-atom : Symbol Atom -> Nat
(define (count-atom sym a)
 (cond
   [(number? a) ... ]
   [(string? a) ... ]
   [(symbol? a) ... ]))
```

```
;; count-ptree : Symbol ProgTree -> Nat
(define (count-ptree sym pt)
  (cond
   [(empty? pt) ...]
   [else  ... (sexpr-fn (first pt)) ... (ptree-fn (rest pt)) ...]))
```

Counting Symbols

```
;; count : Symbol Sexpr -> Nat
;; Computes the number of times the given
;; symbol appears in the given s-expression
(define (count sym se)
  (cond
   [(atom? s) (count-atom sym se)]
   [else (count-ptree sym se)]))
```

```
;; count-atom : Symbol Atom -> Nat
(define (count-atom sym a)
 (cond
   [(number? a) ... ]
   [(string? a) ... ]
   [(symbol? a) ... ]))
```

```
;; count-ptree : Symbol ProgTree -> Nat
(define (count-ptree sym pt)
  (cond
   [(empty? pt) ...]
   [else  ... (sexpr-fn (first pt)) ... (ptree-fn (rest pt)) ...]))
```

Counting Symbols

```
;; count : Symbol Sexpr -> Nat
;; Computes the number of times the given
;; symbol appears in the given s-expression
(define (count sym se)
  (cond
   [(atom? s) (count-atom sym se)]
   [else (count-ptree sym se)]))
```

```
;; count-atom : Symbol Atom -> Nat
(define (count-atom sym a)
 (cond
   [(symbol? a)
    (if (symbol=? sym a) 1 0)]
   [else 0]))
```

```
;; count-ptree : Symbol ProgTree -> Nat
(define (count-ptree sym pt)
  (cond
   [(empty? pt) ...]
   [else  ... (sexpr-fn (first pt)) ... (ptree-fn (rest pt)) ...]))
```

Counting Symbols

```
;; count : Symbol Sexpr -> Nat
;; Computes the number of times the given
;; symbol appears in the given s-expression
(define (count sym se)
  (cond
   [(atom? s) (count-atom sym se)]
   [else (count-ptree sym se)]))
```

```
;; count-atom : Symbol Atom -> Nat
(define (count-atom sym a)
 (cond
   [(symbol? a)
    (if (symbol=? sym a) 1 0)]
   [else 0]))
```

```
;; count-ptree : Symbol ProgTree -> Nat
(define (count-ptree sym pt)
  (cond
   [(empty? pt) 0]
   [else  ... (sexpr-fn (first pt)) ... (ptree-fn (rest pt)) ...]))
```

Counting Symbols

```
;; count : Symbol Sexpr -> Nat
;; Computes the number of times the given
;; symbol appears in the given s-expression
(define (count sym se)
  (cond
   [(atom? s) (count-atom sym se)]
   [else (count-ptree sym se)]))
```

```
;; count-atom : Symbol Atom -> Nat
(define (count-atom sym a)
 (cond
   [(symbol? a)
    (if (symbol=? sym a) 1 0)]
   [else 0]))
```

```
;; count-ptree : Symbol ProgTree -> Nat
(define (count-ptree sym pt)
  (cond
   [(empty? pt) 0]
   [else (+ (count sym (first pt))
            (count-ptree sym (rest pt)))]))
```

Counting Symbols

```
;; count : Symbol Sexpr -> Nat
;; Computes the number of times the given
;; symbol appears in the given s-expression
(define (count sym se)
  (cond
   [(atom? s) (count-atom sym se)]
   [else (count-ptree sym se)]))
```

```
;; count-atom : Symbol Atom -> Nat
(define (count-atom sym a)
 (cond
   [(symbol? a)
    (if (symbol=? sym a) 1 0)]
   [else 0]))
```

```
;; count-ptree : Symbol ProgTree -> Nat
(define (count-ptree sym pt)
  (cond
   [(empty? pt) 0]
   [else (+ (count sym (first pt))
            (count-ptree sym (rest pt)))]))
```

A “Racket Prog” = S-expression!

```
;; A ~~RacketProg~~ Sexpr is one of:
;; - Atom
;; - ProgTree
```

```
(define (sexpr-fn s)
  (cond
   [(atom? s) ... (atom-fn s) ...]
   [else  ...
```

```
;; An Atom is one of:
;; - Number
;; - String
;; - Symbol
```

```
(define (atom-fn a)

     mber? a) ... ]
     ring? a) ... ]
   [(symbol? a) ... ]))
```

An **S-expression** is the **syntax** of a **Racket program**

```
;; A ProgTree
;; - empty
;; - (cons ~~RacketProg~~ Sexpr ProgTree)
```

```
(define (ptree-fn t)
  (cond
   [(empty? t) ...]
   [else  ... (sexpr-fn (first t)) ... (ptree-fn (rest t)) ...]))
```