UMass Boston Computer Science

CS450 **High Level Languages**

Tree Data Definitions,

and accumulators

Tuesday, April 1, 2025

Logistics

- HW 7 in
  - due: Tues 4/1, 11 am EST
- HW 8 out (extra credit)
  - due: Tues 4/8, 11 am EST
  - 3 options:
    - Continue working
    - Revise a previous
    - Complete the gam

Logistics

- HW 7 in
  - ~~<u>due:</u> Tues 4/1, 11 am EST~~

- **HW 8 out** (extra credit)
  - <u>due:</u> Tues 4/8, 11 am EST

  3 options:
  - Continue working on hw7
  - Revise a previous assignment
  - Complete the game

Last Time

More Recursive Data Definitions: Trees

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

Last Time

Tree Template

```
;; A Tree<X> is one of:
;; - empty
;; - (node Tree<X> X Tree<X>)
(struct node [left data right])
;; 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)) ...]))
```

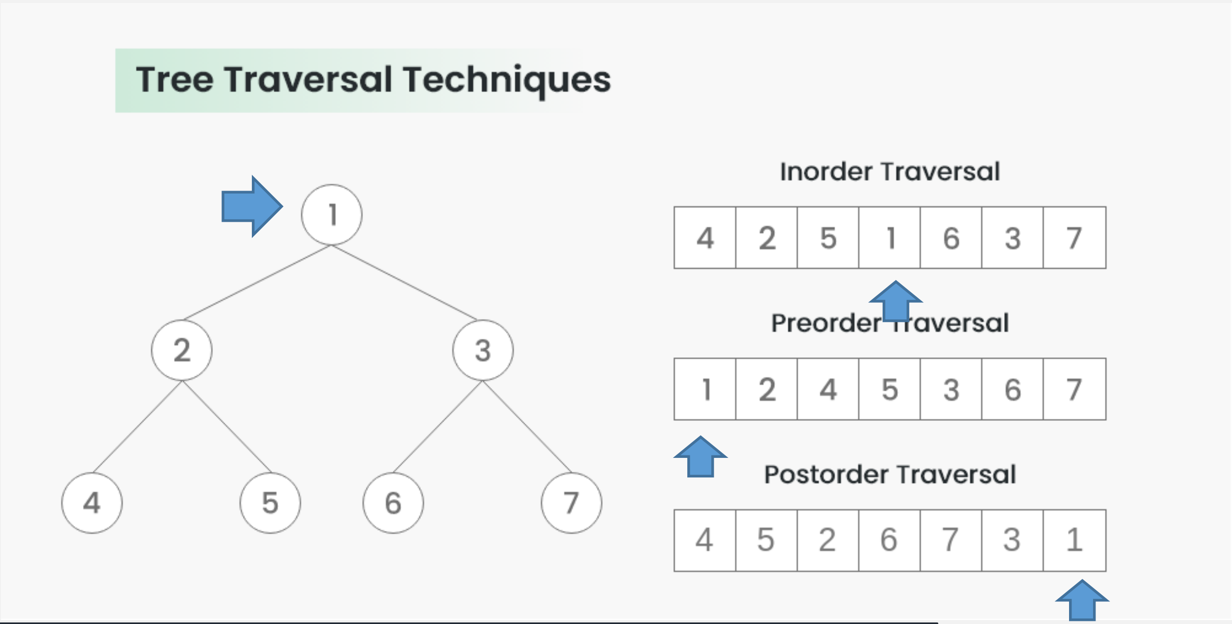
Template:
Recursive call(s) match recursion in data definition

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
```

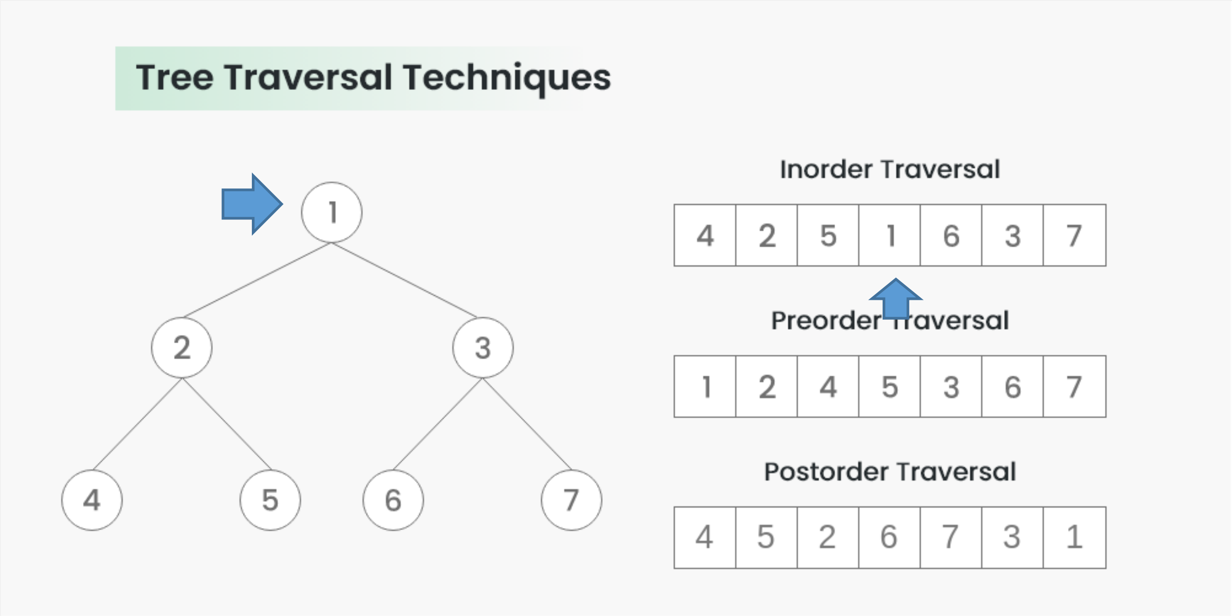
```
;; tree->lst/pre : Tree<X> -> List<X>
;; converts given tree to a list of values, by preorder
```

```
;; 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



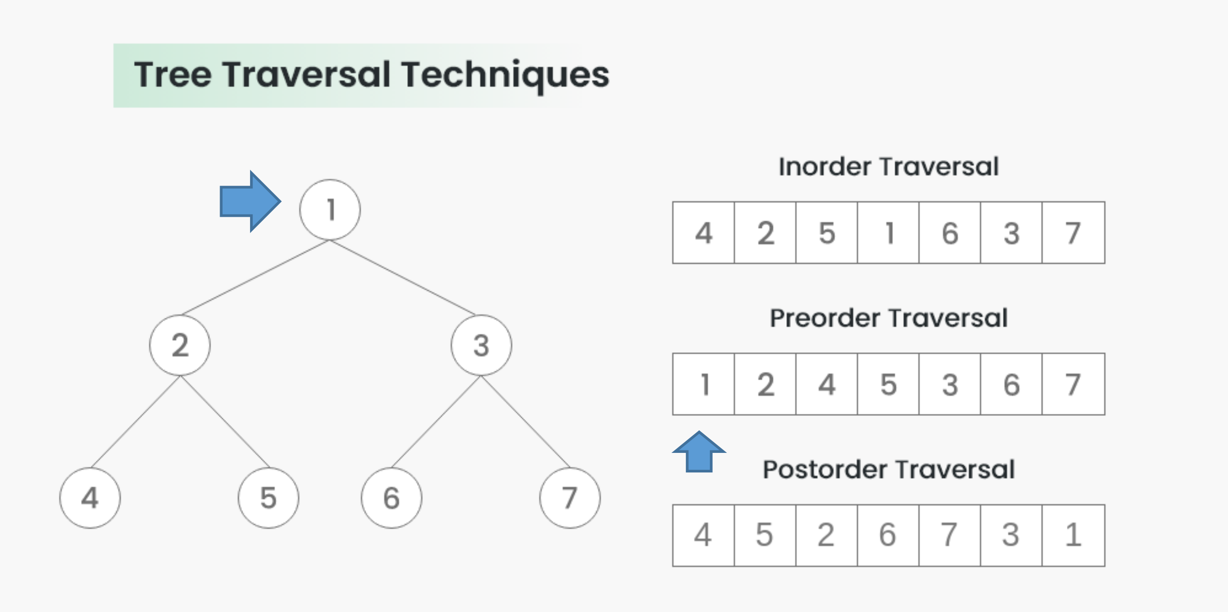

```
;; tree->lst/in : Tree<X> -> List<X>
;; converts given tree to a list of values, by inorder
```

```
(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))))]))
```

Must figure out how to "combine pieces"

Last Time

Pre-order Traversal



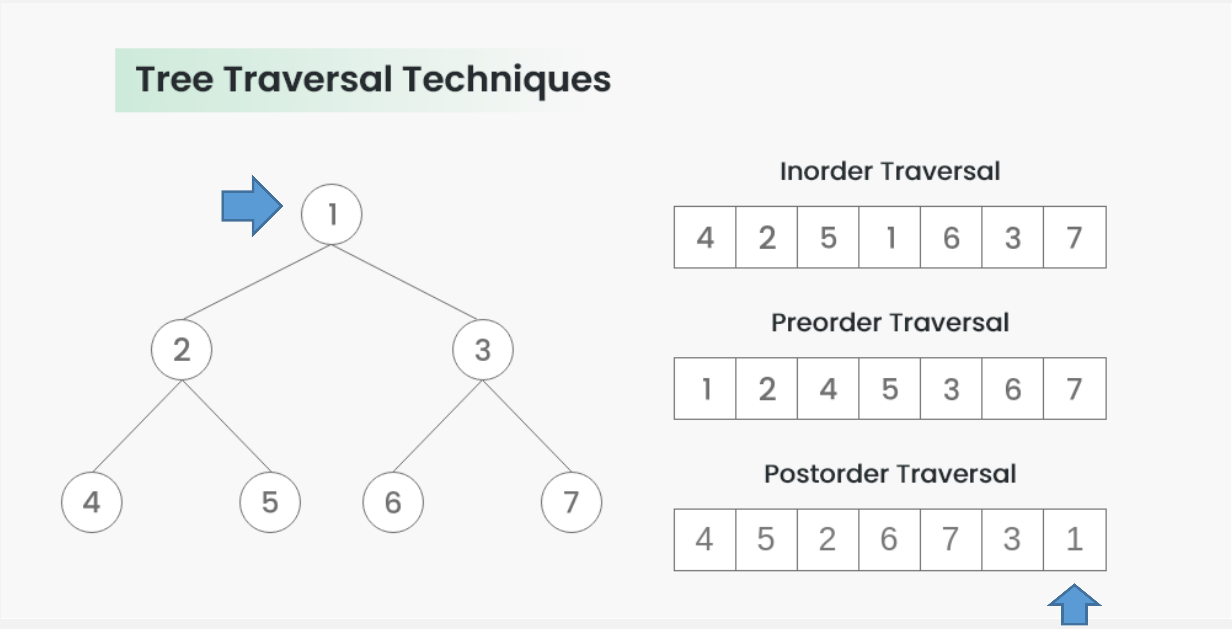

```
;; tree->lst/pre : Tree<X> -> List<X>
;; converts given tree to a list of values, by preorder
```

```
(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))))]))
```

Must 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)))]))
```

Must figure out how to "combine pieces"

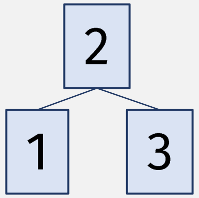
Last Time

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? (curryr < 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

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
;; - (node Tree<X> X Tree<X>)
(struct node [left data right])
;; a binary tree data structure
```

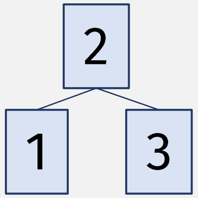
```
;; A BinarySearchTree<X> (BST) is a Tree<X>
;; where, if tree is a node:
;; Invariant 1: ∀x ∈left tree, x < node-data
;; Invariant 2: ∀y ∈right tree, y ≥ node-data
;; Invariant 3: left subtree must be a BST
;; Invariant 4: right subtree must be a BST
```

(deep) predicate?

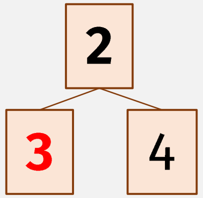
Valid BSTs

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

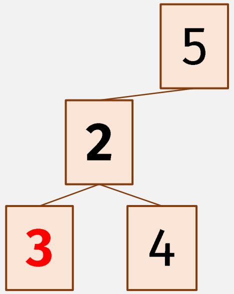
Valid



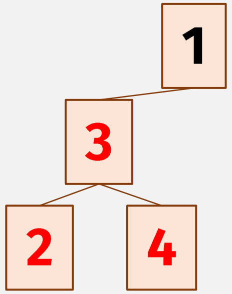
Not Valid



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

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


                         (node-left t))
           (valid-bst/p? ???


                         (node-right
```

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

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

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


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

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



                         (node-right
```

new “**p?**”

Need to still check previous **p?**

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

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

new "**p?**"

Need to still check previous p

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

```
(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)))))
```

```
(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**

In-class Coding: Tree Max

Accumulator used for "remembering" info, but doesn't always "accumulate"

```
;; tree-max : TreeNode<Int> -> Int
;; Returns the maximum value in a given (non-empty) (non-BST) tree
(define (tree-max t0)

  ;; tree-max/a : Tree<Int> -> Int
  ;; accumulator root-val: Int
  ;; invariant: node-data of t0 root node (max of empty tree)
  (define (tree-max/a t root-val)
   (cond
     [(empty? t) root-val]
     [else (max (node-data t)
                (tree-max/a (node-left t) root-val)
                (tree-max/a (node-right t) root-val))]))

  (tree-max/a t0 (node-data t0)))
```

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

(need a "default" max for empty tree)

2. *Define* "helper" fn with **accumulator** (and other args)

This accum doesn't change

This is not the only possible accumulator choice

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

In-class Coding: Tree Max #2

```
;; tree-max : TreeNode<Int> -> Int
;; Returns the maximum value in a given (non-empty) (non-BST) tree
(define (tree-max t0)

  ;; tree-max/a : Tree<Int> -> Int
  ;; accumulator root-val: Int
  ;; invariant: node-data of root parent node (max of empty tree)
  (define (tree-max/a t root-val parent-val)
   (cond
     [(empty? t) root-val parent-val]
     [else (max (node-data t) parent-val
                (tree-max/a (node-left t) root-val (node-data t))
                (tree-max/a (node-right t) root-val (node-data t)))]))

  (tree-max/a t0 (node-data t0)))
```

(need a "default" max for empty tree)

Pass **node-data** of parent on recursive call

The **accumulator invariant** is key to understanding the program!

Last Time

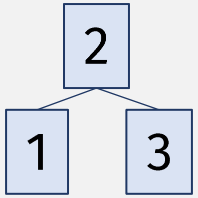
BST Insert

Must preserve BST invariants

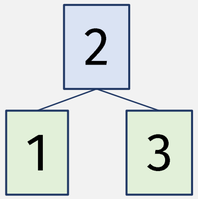
Hint: use **valid-bst?** For tests

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

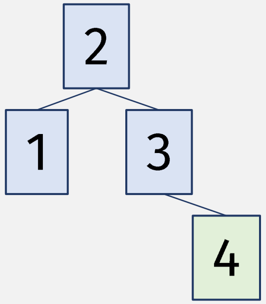
```
(define TREE2 (node empty 2 empty))
(define TREE123 (node TREE1 2 TREE3))
```




```
(check-equal? (bst-insert (bst-insert TREE2 1) 3)
              TREE123))
```




```
(check-true (valid-bst? (bst-insert TREE123 4)))
```



BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< x (node-data bst))
         (node (bst-insert (node-left bst) x)
               (node-data bst)
               (node-right bst))
         (node (node-left bst)
               (node-data bst)
               (bst-insert (node-right bst) x)))]))
```

Template:
cond clause for **each** itemization item

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< x (node-data bst))
         (node (bst-insert (node-left bst) x)
               (node-data bst)
               (node-right bst))
         (node (node-left bst)
               (node-data bst)
               (bst-insert (node-right bst) x)))]))
```

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< x (node-data bst))
         (node (bst-insert (node-left bst) x)
               (node-data bst)
               (node-right bst))
         (node (node-left bst)
               (node-data bst)
               (bst-insert (node-right bst) x)))]))
```

Template:
Recursive call matches recursion in data definition

Template:
Extract pieces of compound data

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< x (node-data bst))
         (node (bst-insert (node-left bst) x)
               (node-data bst)
               (node-right bst))
         (node (node-left bst)
               (node-data bst)
               (bst-insert (node-right bst) x)))]))
```

Result must maintain **BST invariant!**

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< x (node-data bst))
         (node (bst-insert (node-left bst) x)
               (node-data bst)
               (node-right bst))
         (node (node-left bst)
               (node-data bst)
               (bst-insert (node-right bst) x)))]))
```

Result must maintain **BST invariant!**

Smaller values on left

BST Insert

```
;; bst-insert : BST<X> X -> BST<X>
;; inserts given val into given bst, result is still a bst
```

```
(define (bst-insert bst x)
  (cond
    [(empty? bst) (node empty x empty)]
    [(node? bst)
     (if (< x (node-data bst))
         (node (bst-insert (node-left bst) x)
               (node-data bst)
               (node-right bst))
         (node (node-left bst)
               (node-data bst)
               (bst-insert (node-right bst) x)))]))
```

Result must maintain **BST invariant!**

Larger values on right

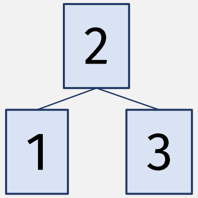
Finding a Value in a Tree?

- Do we have to search the entire tree?

Finding a Value in a BST?

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

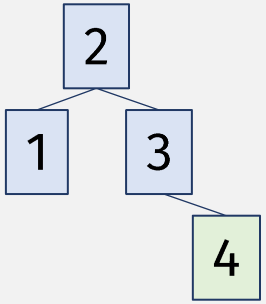
```
(define TREE1 (node empty 1 empty))
(define TREE3 (node empty 3 empty))
(define TREE123 (node TREE1 2 TREE3))
```




```
(check-true (valid-bst? TREE123))
```

```
(check-true (bst-has? TREE123 1))
(check-false (bst-has? TREE123 4))
```

```
(check-true (bst-has? (bst-insert TREE123 4) 4))
```



Finding a Value in a BST?

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

```
(define (bst-has? bst x)
        ???  (empty? bst)
        ???              (node-data bst)
        ??? (bst-has? (node-left bst) x)
        ??? (bst-has? (node-right bst) x) )
```

BST (bool result) **Template**

Finding a Value in a BST?

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       ???            (node-data bst)
       ??? (bst-has? (node-left bst) x)
       ??? (bst-has? (node-right bst) x) )
```

BST cannot be empty

Finding a Value in a BST?

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       (or (equal? x (node-data bst))
       ??? (bst-has? (node-left bst) x)
       ??? (bst-has? (node-right bst) x) )
```

Either:
- (node-data bst) is x

Finding a Value in a BST?

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       (or (equal? x (node-data bst))
           (bst-has? (node-left bst) x)
       ??? (bst-has? (node-right bst) x) )
```

Either:
- (node-data bst) is x
- left subtree has x

What about BST invariants?

Should never have to check both trees

Finding a Value in a BST?

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       (or (equal? x (node-data bst))
           (if (< x (node-data bst))
               (bst-has? (node-left bst) x)
               (bst-has? (node-right bst) x)))))
```

Either:
- (node-data bst) is x
- left subtree has x (if **x** < **data**)
- right subtree has x (if **x** > **data**)

Should never have to check both trees

Finding a Value in a BST?

```
;; bst-has?: BST<X> X -> Bool
;; Returns true if the given BST has the given value
```

```
(define (bst-has? bst x)
  (and (not (empty? bst))
       (or (equal? x (node-data bst))
           (if (< x (node-data bst))
               (bst-has? (node-left bst) x)
               (bst-has? (node-right bst) x)))))
```

and and **or** are "short circuiting"
(stop search as soon as **x** is found)

Intertwined Data Definitions

- Come up with **a Data Definition** for **…**

- **… valid Racket Programs**

Valid Racket Programs

- 1
- “one”
- (+ 1 2)

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

Tree?

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

But: how many values does each node have??

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

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

Valid Racket Programs

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

Tree?

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 set of Data Definitions that reference each other
- Templates 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 set of Data Definitions that reference each other
- Templates 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) ...)
```

???

Submit: to GradeScope

In-class Coding #2: Intertwined Templates

- Templates 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) ... ]
   [else ... ]))
```

```
;; 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) ...]))
```

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

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

```
;; 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)) ...]))
```