UMass Boston Computer Science

CS450 High Level Languages (section 2)

Tree Data Definitions,
and accumulators

Monday, October 28, 2024

Logistics

- HW 7 in
  - <u>due:</u> Mon 10/28 12pm (noon) EDT

- HW 8 out
  - <u>due:</u> Mon 11/4 12pm (noon) EDT

- No class Mon 11/11 (Veteran's Day)

Racket **for** expressions

Generic "sequence"
(number, most data structures ...)

```
(for/list ([x lst]) (add1 x))
```

```
(map add1 lst)
```

```
(for/list ([x n]) (add1 x))
```

```
(build-list n add1)
```

```
(for/list ([x lst] #:when (odd? x)) (add1 x))
```

```
(filter odd? (map add1 lst))
```

Note:
These are still expressions!

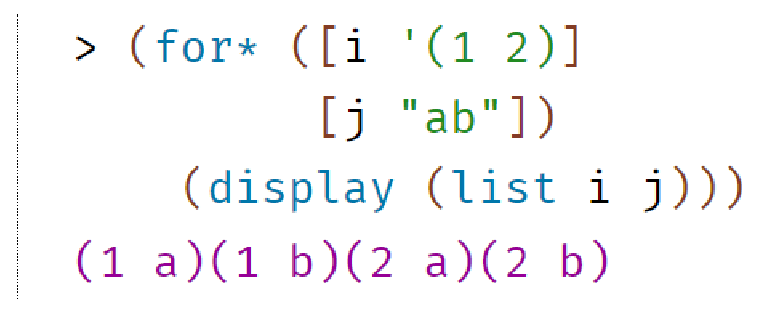
```
(for/sum ([x lst] #:when (odd? x)) (add1 x))
```

```
(foldl + 0 (filter odd? (map add1 lst)))
```

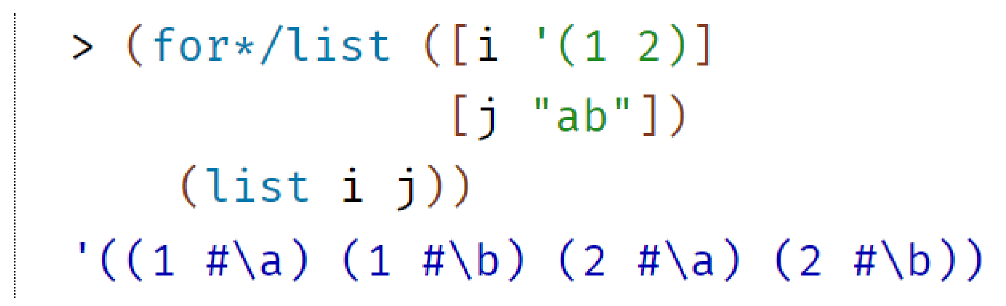
Lots of variations!
(see docs)

Racket **for*** expressions

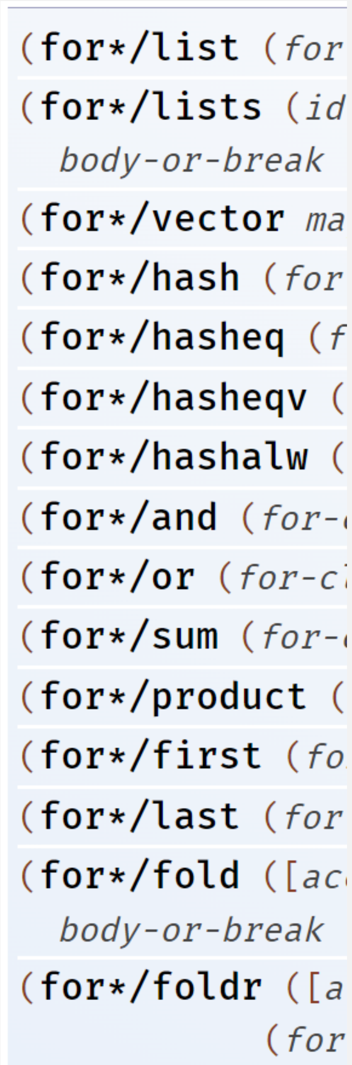
“nested” for loops



```
> (for* ([i '(1 2)]
         [j "ab"])
    (display (list i j)))
(1 a)(1 b)(2 a)(2 b)
```



```
> (for*/list ([i '(1 2)]
              [j "ab"])
    (list i j))
'((1 #\a) (1 #\b) (2 #\a) (2 #\b))
```

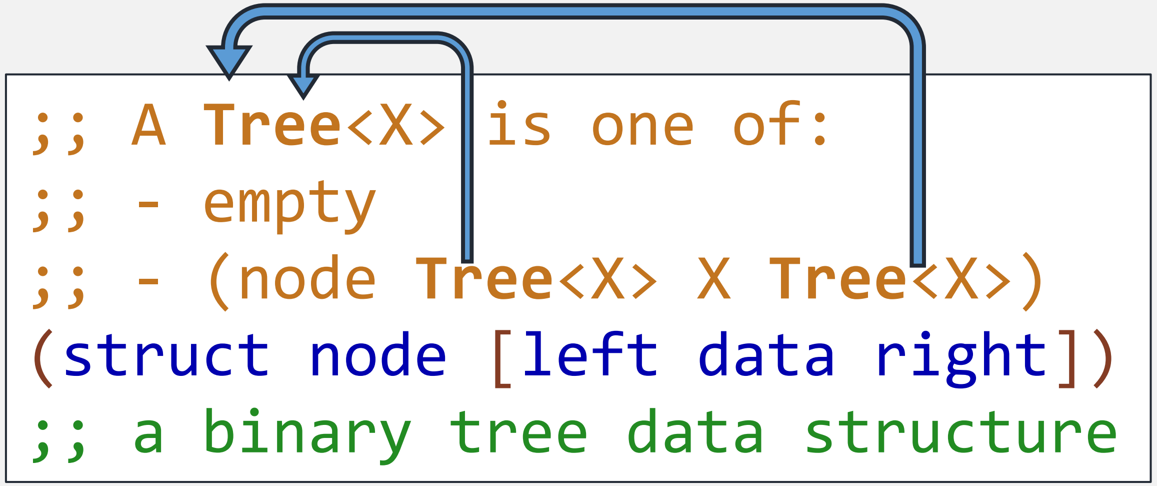


```
(for*/list (for
(for*/lists (id
  body-or-break
(for*/vector ma
(for*/hash (for
(for*/hasheq (f
(for*/hasheqv (
(for*/hashalw (
(for*/and (for-
(for*/or (for-c
(for*/sum (for-
(for*/product (
(for*/first (fo
(for*/last (for
(for*/fold ([ac
  body-or-break
(for*/foldr ([a
           (for
```

Lots of variations! (see docs)

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

In-class Coding: 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)) ...]))
```

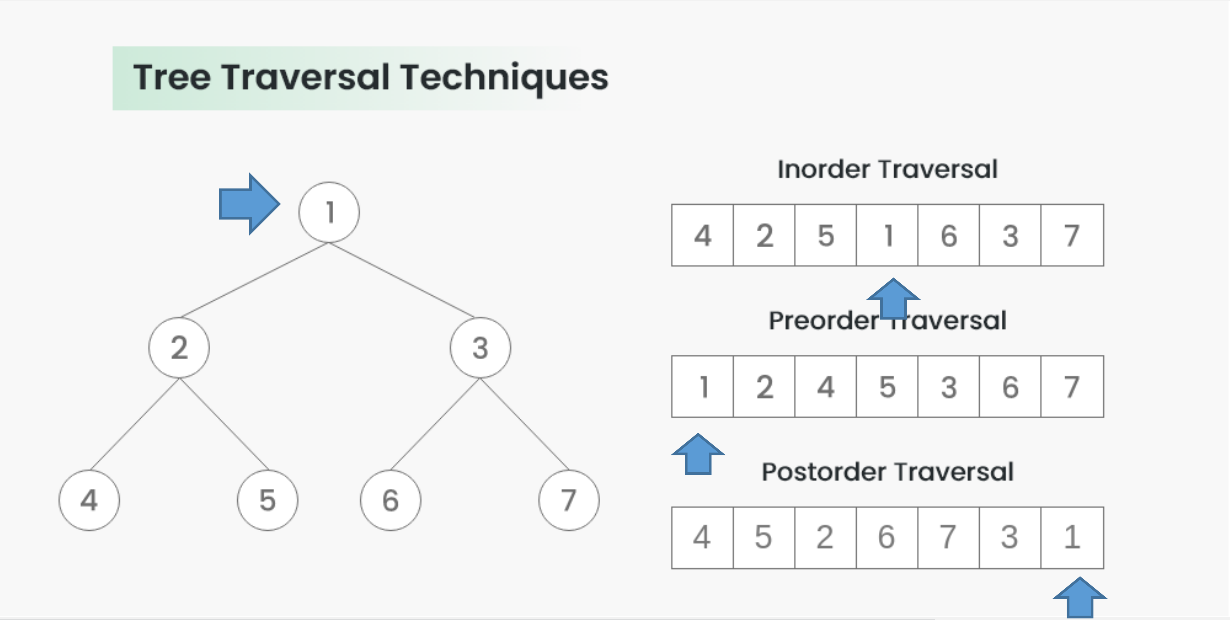
Template:
cond clause for each itemization item

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

Template:
Extract pieces of compound data

Last Time

Tree Algorithms



Tree Traversal Techniques

Inorder Traversal

4	2	5	1	6	3	7

Preorder Traversal

1	2	4	5	3	6	7

Postorder Traversal

4	5	2	6	7	3	1

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

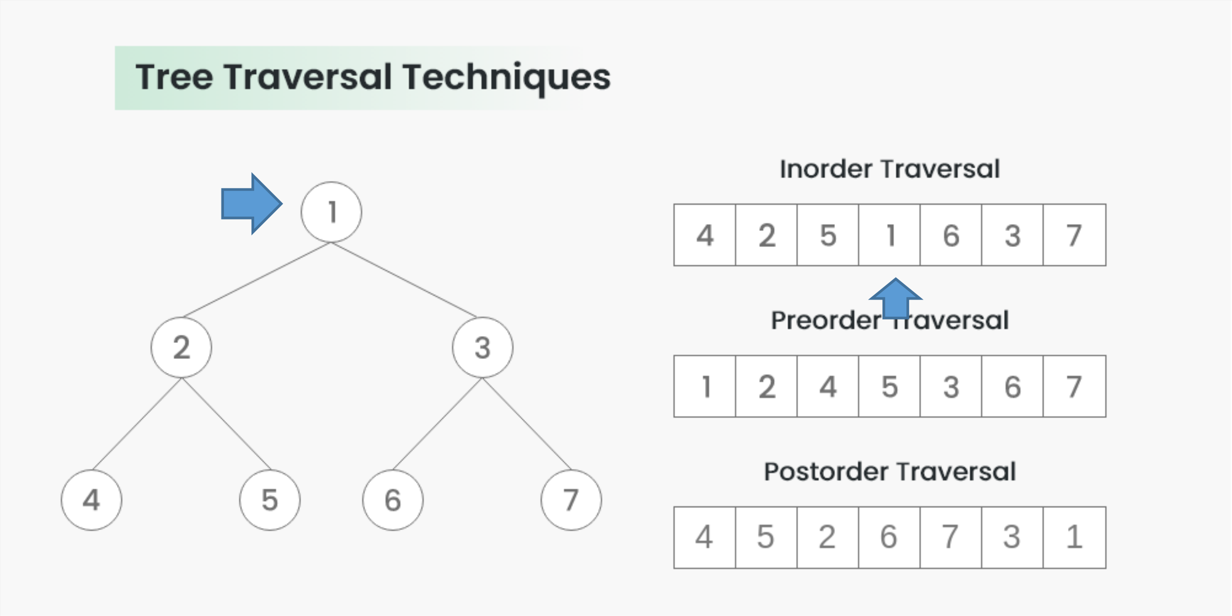
```
;; tree->lst/pre : Tree<X> -> List<X>
;; converts given tree to a list of values, by preorder
```

```
;; 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

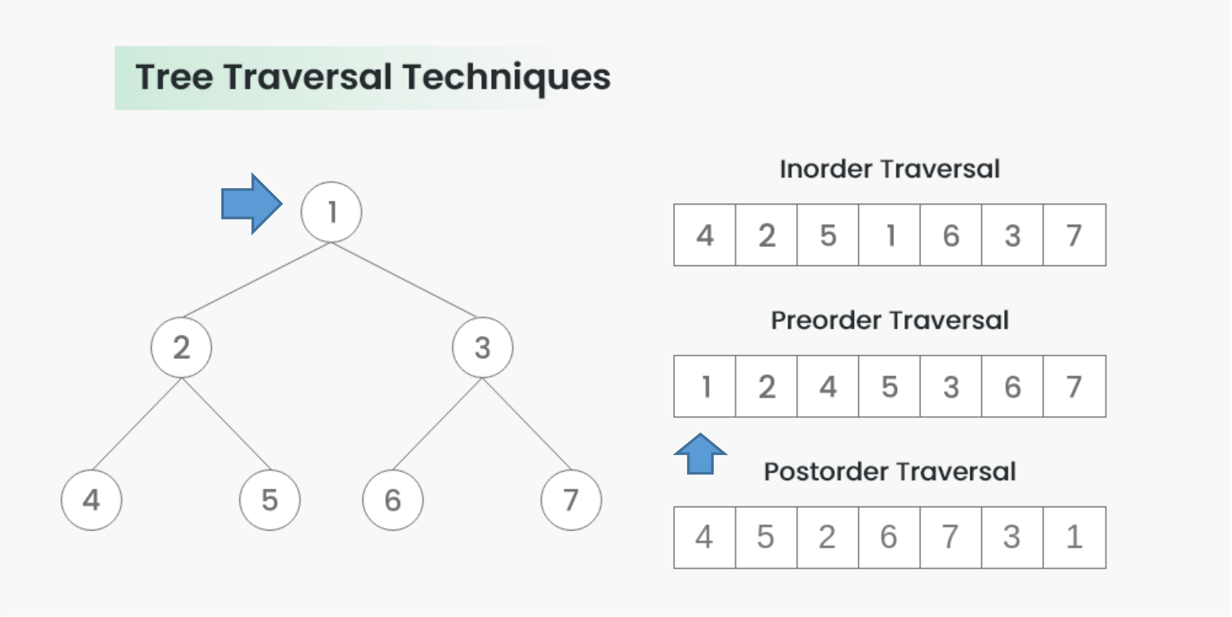


```
;; tree->lst/in : Tree<X> -> List<X>
;; converts given tree to a list of values, by inorder
```

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

Last Time

Pre-order Traversal

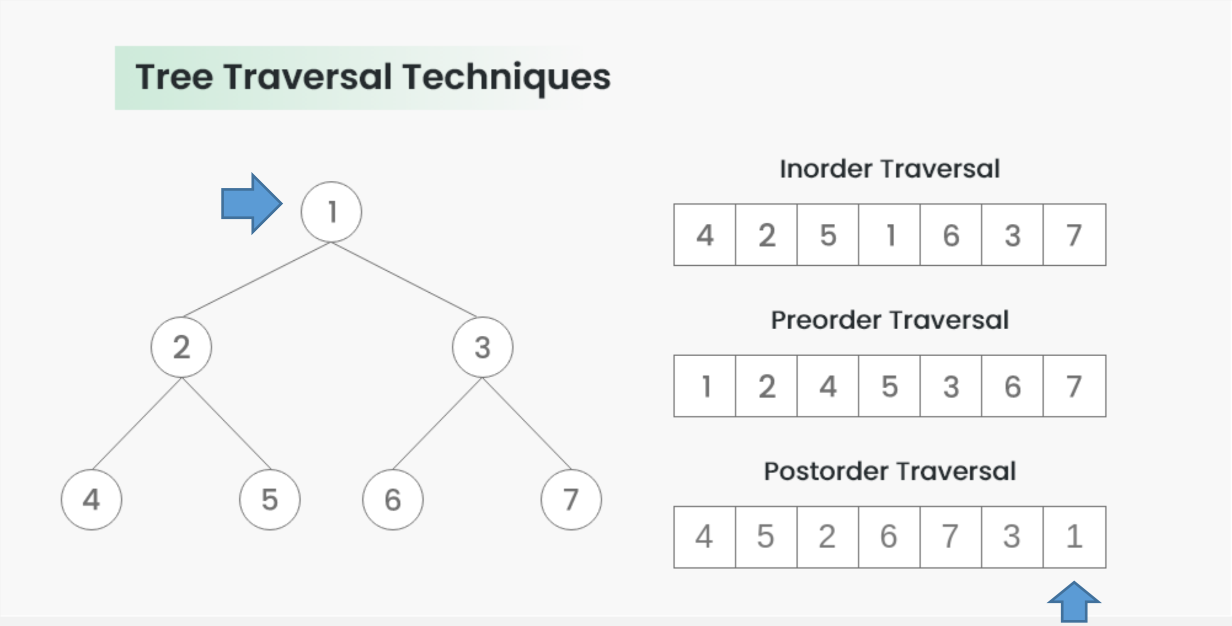


```
;; tree->lst/pre : Tree<X> -> List<X>
;; converts given tree to a list of values, by preorder
```

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

Last Time

Post-order Traversal



Tree Traversal Techniques

Inorder Traversal

4	2	5	1	6	3	7

Preorder Traversal

1	2	4	5	3	6	7

Postorder Traversal

4	5	2	6	7	3	1

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

Last Time

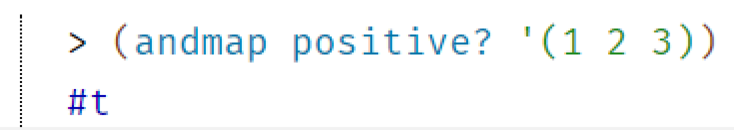
tree-all?

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

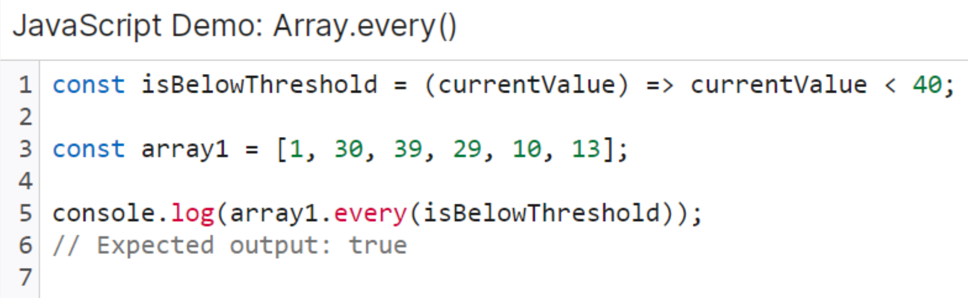
```
(define TREE1 (node empty 1 empty))
(define TREE3 (node empty 3 empty))
(define TREE123 (node TREE1 2 TREE3))
```

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

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



```
> (andmap positive? '(1 2 3))
#t
```



JavaScript Demo: Array.every()

```
const isBelowThreshold = (currentValue) => currentValue < 40;

const array1 = [1, 30, 39, 29, 10, 13];

console.log(array1.every(isBelowThreshold));
// Expected output: true

```

Last Time

tree-all?

```
;; tree-all? : (X -> Boolean) Tree<X> -> Boolean
;; Returns true if given pred 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 pred 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 pred 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 pred 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 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
```

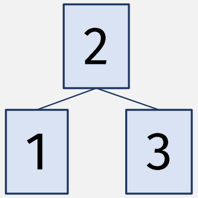
Predicate?

```
;; 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
```

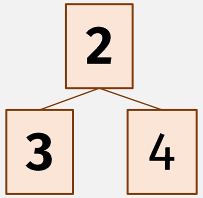
Valid BSTs

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

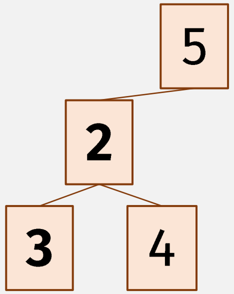
Valid



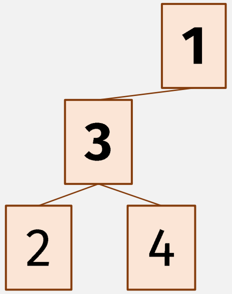
Not Valid



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 ☒

In-class Coding #1: Valid BST

Hint: `use tree-all?`

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

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

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

```
(check-false (valid-bst? (node TREE3 1 TREE2)))
```

Remember:
boolean arithmetic doesn’t use cond

- `git clone git@github.com:`**`cs450f24/in-class-10-28`**
- `git add` **`bst-valid`**`-<Last>-<First>.rkt`
  - E.g., `bst-valid-Chang-Stephen.rkt`
- `git commit bst-valid-Chang-Stephen.rkt –m ‘add Chang valid-bst?’`
- `git push origin main`
- Might need: `git pull --rebase`
  - If your local clone is not at HEAD

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

Valid BSTs

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

```
;; 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
```

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

cond that evaluates to a boolean 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)))))
```

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)))))
```

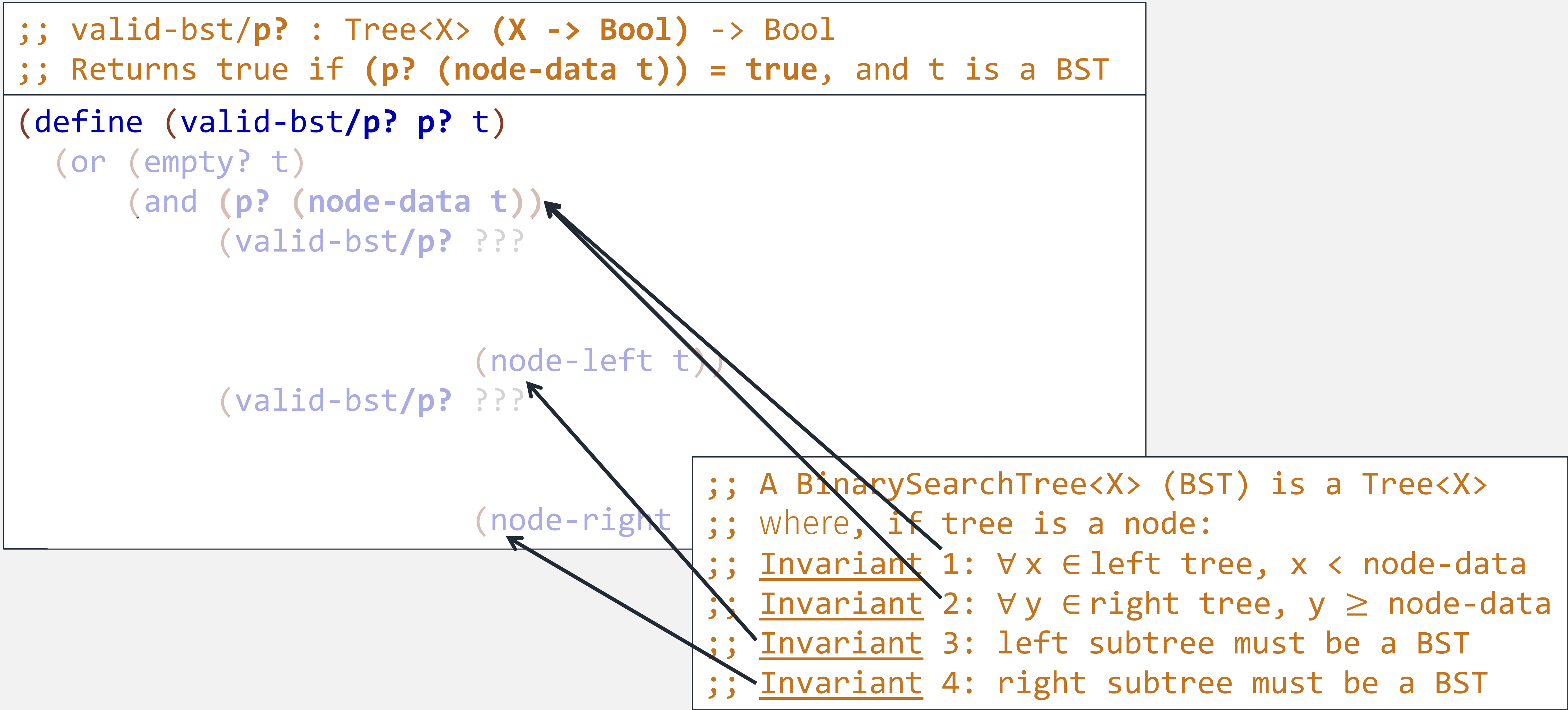
One-pass **valid-bst?**

```
;; 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 valid interval for node-data value

One-pass **valid-bst?**, Functional 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? ???


                         (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?**, Functional 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?

                                 (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?**, Functional 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? (lambda (x)
                           (and (p? x)
                                ((curry > (node-data t)) x)))
                         (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?**, Functional 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? (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)))))
```

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

“conjoin”
combines
predicates

One-pass **`valid-bst?`**, Functional 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)))))
```

```
      (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?**, Functional 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
                         (node-left t))
           (valid-bst/p? (conjoin p? (curry <= (node-dat
                         (node-right t)))))
```

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

```
;; A BinarySearchTree<X> is a Tree
;; where, if tree is a node:
;; Inv1: ∀ x ∈ left, x < node-data
;; Inv2: ∀ y ∈ right, y ≥ node-data
;; Inv3: left subtree must be BST
;; Inv4: right subtree must be BST
```

"Extra argument" is called an **accumulator**

"conjunction" = AND

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

Last Time

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

In-class Coding #1: Valid BST – with accum

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: ???
```

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

- `git clone git@github.com:`**`cs450f24/in-class-10-28`**
- `git add` **`bst-valid2`**`-<Last>-<First>.rkt`
    - E.g., `bst-valid2-Chang-Stephen.rkt`
- `git commit bst-valid2-Chang-Stephen.rkt –m ‘add Chang valid-bst? accum’`
- `git push origin main`
- Might need: `git pull --rebase`
    - If your local clone is not at HEAD

2. *Define* internal “helper” fn with **accumulator** arg

3. *Call* “helper” fn, with initial **accumulator**

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? remembers valid vals
  ;; for node-data such that (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**

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 (< (node-data bst))
         (node (bst-insert (node-left t) x)
               (node-data t)
               (node-right t))
         (node (node-left t)
               (node-data t)
               (bst-insert (node-right t) 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 (< (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 (< (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 (< (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

Last Time

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))
```

In-class Coding #3: BST-has?

```
;; 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
```

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

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

- git add **bst-has-**<Last>-<First>.rkt
  - E.g., bst-has-Chang-Stephen.rkt
- git commit bst-has-Chang-Stephen.rkt
  –m ‘add chang bst-has?’
- git push origin main
- Might need: git pull --rebase
  - If your local clone is not at HEAD

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

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

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
- right subtree has 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))))
```

`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
```

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
;; - 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
```

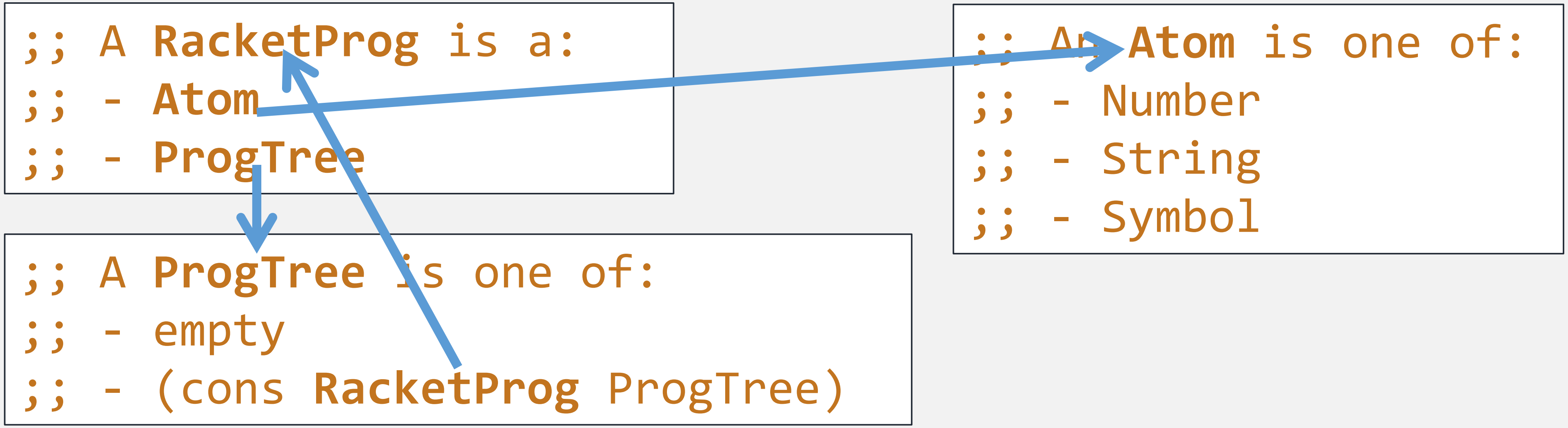
```
;; An Atom is a:
;; - Number
;; - String
;; - Symbol
```

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

Recursive Data Def!

Valid Racket Programs

Also, **Intertwined** Data Defs!



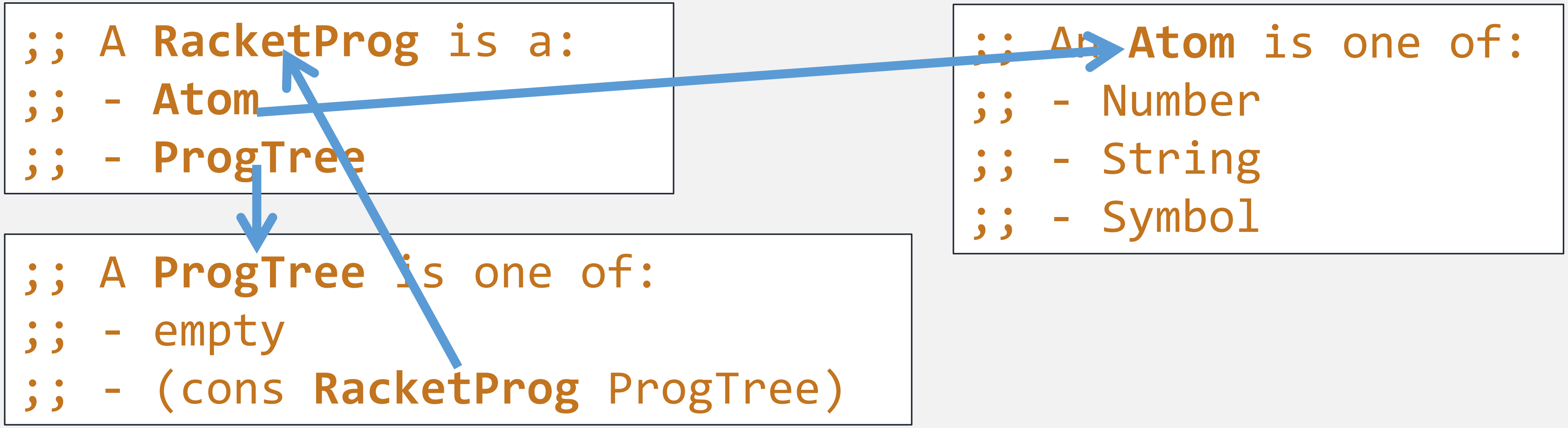
```
;; A RacketProg is a:
;; - Atom
;; - ProgTree
```

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

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

Intertwined Data

- A set of Data Definitions that reference each other
- Templates should be defined together …



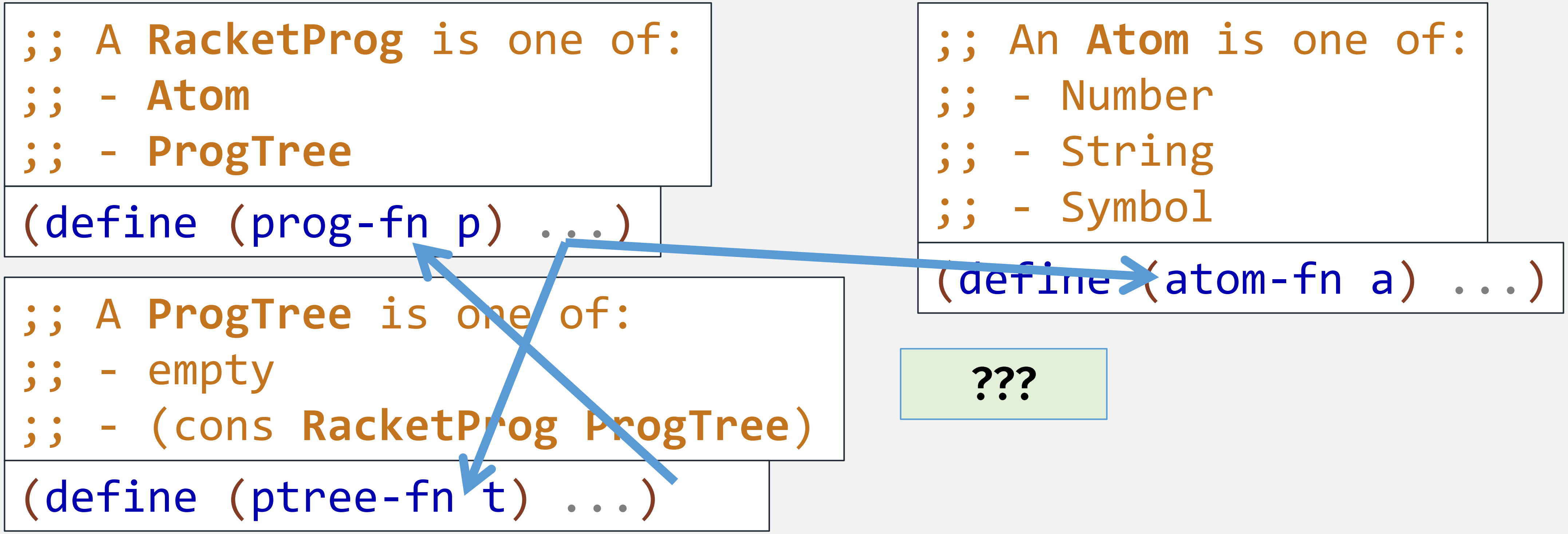
```
;; A RacketProg is a:
;; - Atom
;; - ProgTree
```

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

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

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) ...)
```

???

- Repo: **cs450f24/in-class-10-28**
- File: **intertwined-template**-<*Last*>-<*First*>.rkt

In-class Coding #4: 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
   [(list? s) ... (ptree-fn s) ...]
   [else       ... (atom-fn s) ...]))
```

Can swap cond ordering
(to make distinguishing
items easier)

```
;; 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!

“Racket Prog” = S-expression!

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

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

```
;; A ProgTree is one of:
;; - empty
;; - (cons 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) ... ]
   [else ... ]))
```