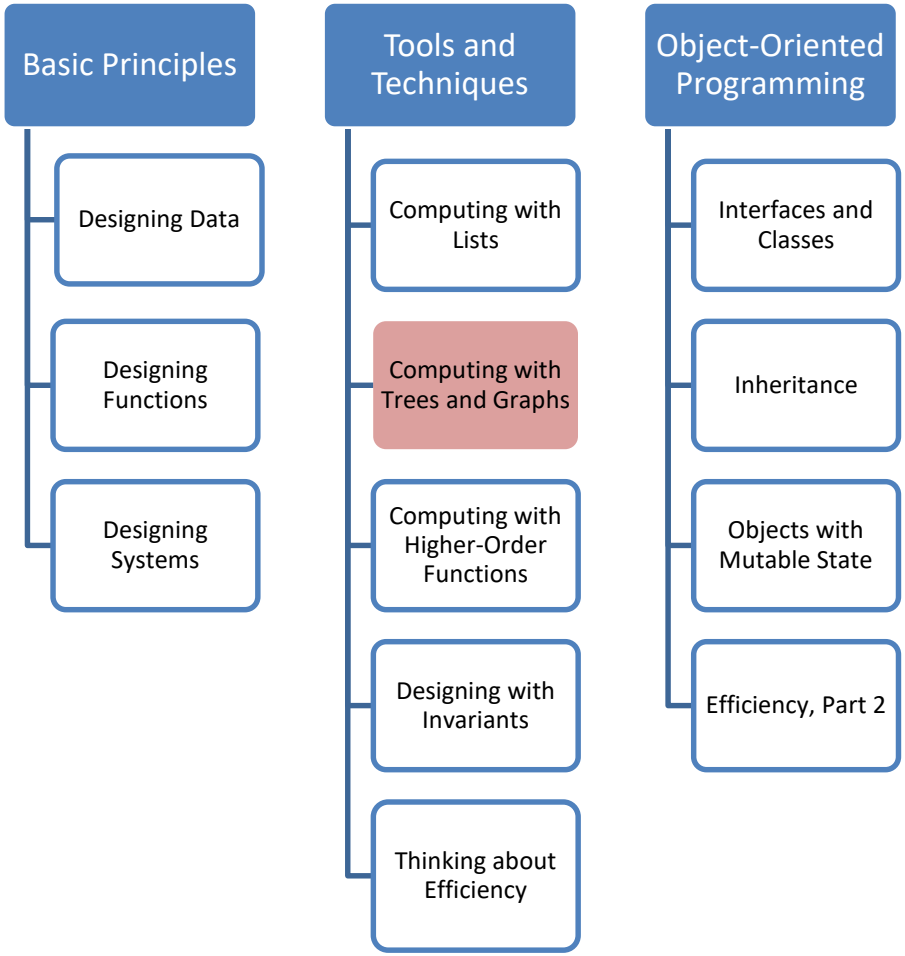
Trees

CS 5010 Program Design Paradigms "Bootcamp"

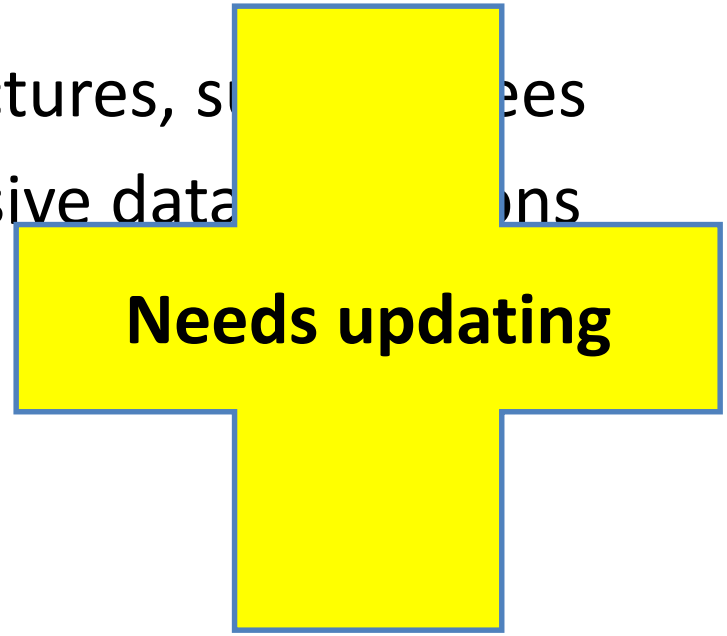
Lesson 5.1

Module 05


Basic Principles
Designing Data
Designing Functions
Designing Systems
Tools and Techniques
Computing with Lists
Computing with Trees and Graphs
Computing with Higher-Order Functions
Designing with Invariants
Thinking about Efficiency
Object-Oriented Programming
Interfaces and Classes
Inheritance
Objects with Mutable State
Efficiency, Part 2

Module Introduction

- In this module we will learn about two related topics:
  - branching structures, s ees
  - mutually recursive data ns



Module Outline

- Lesson 6.1 begins by considering alternative representations for sequence information
  - This is a warm-up for Lessons 6.2-6.3
- Lessons 6.2 and 6.3 show ho sent information that has a naturally branchin , such as trees
- Lesson 6.4 introduces mutua ve data definitions
- Lesson 6.5 applies th
  - S-expressions are ne
  - These are the basis for XML a
- Lesson 6.6 combines all these o a case study
- Lesson 6.7 shows how to writ measures for tree-like structures.

Lesson Introduction

- Many examples of information have a natural structure which is not a sequence, but is rather a tree, which you should have learned about in your data structures course.
- In this lesson, we'll study how to apply the Design Recipe to trees.

Learning Objectives

- At the end of this lesson you should be able to:
  - Write a data definition for tree-structured information
  - Write functions that manipulate that data, using the observer template

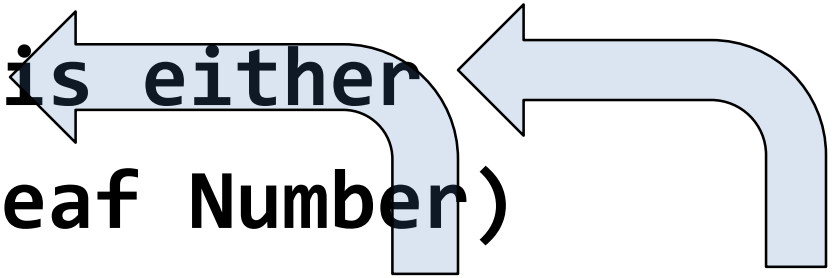
Binary Trees: Data Definition

```
;; A Binary Tree is represented as a BinTree, which is either:
;; (make-leaf datum)
;; (make-node lson rson)

;; INTERPRETATON:
;; datum      : Real       some real data
;; lson, rson : BinTree    the left and right sons of this node

;; IMPLEMENTATION:
(define-struct leaf (datum))
(define-struct node (lson rson))

;; CONSTRUCTOR TEMPLATES:
;; -- (make-leaf Number)
;; -- (make-node BinTree BinTree)
```

There are many ways to define binary trees. We choose this one because it is clear and simple.

Observer Template to follow...

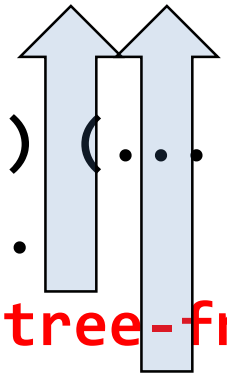
This definition is self-referential (recursive)



```
;; A BinTree is either
;; -- (make-leaf Number)
;; -- (make-node BinTree BinTree)
```

Observer Template

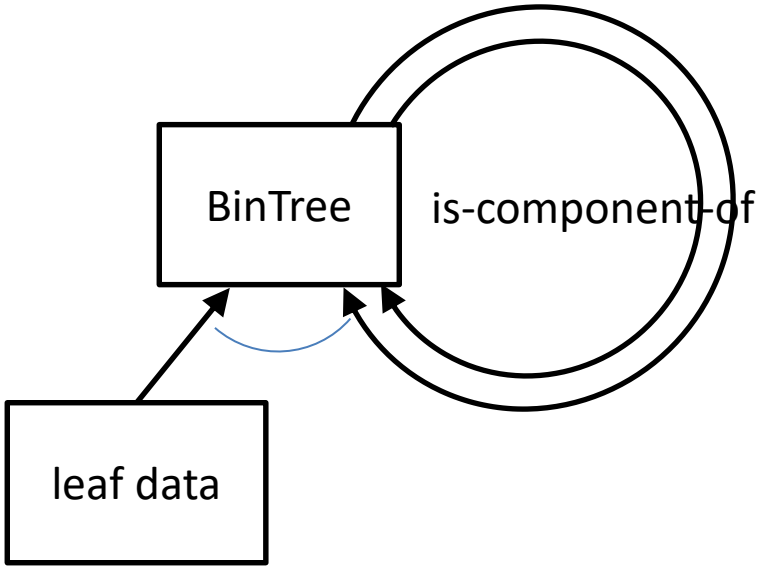
```
tree-fn : BinTree -> ???
(define (tree-fn t)
  (cond
    [(leaf? t) (... (leaf-datum t))]
    [else (...
           (tree-fn (node-lson t))
           (tree-fn (node-rson t)))]))
```



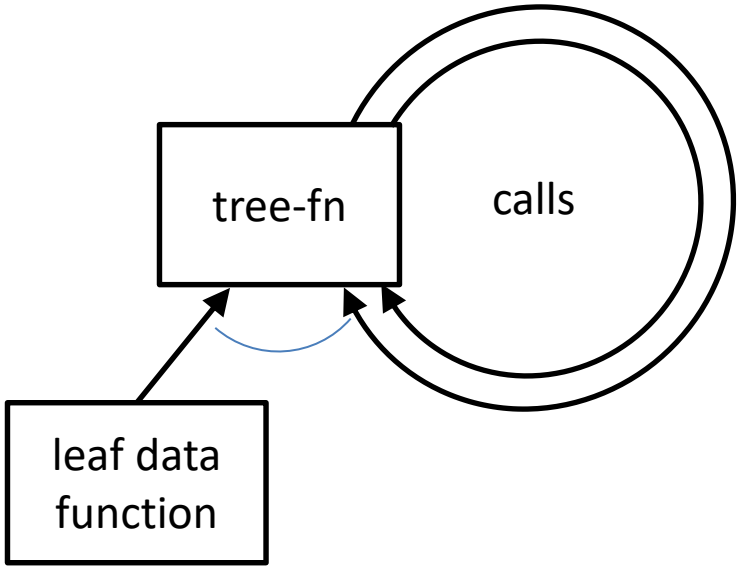
Here's the template for this data definition. Observe that we have two self-references in the template, corresponding to the two self-references in the data definition.

Self-reference in the data definition leads to self-reference in the template; Self-reference in the template leads to self-reference in the code.

Remember: The Shape of the Program Follows the Shape of the Data



Data Hierarchy (a **BinTree** is either leaf data or has two components which are **BinTrees**



Call Tree (**tree-fn** either calls a function on the leaf data, or it calls itself twice.)

The template questions

```
tree-fn : Tree -> ???
(define (tree-fn t)
  (cond
    [(leaf? t) (... (leaf-datum t))]
    [else (...
           (tree-fn (node-lson t))
           (tree-fn (node-rson t) ))]))
```

What's the answer for a leaf?

If you knew the answers for the 2 sons, how could you find the answer for the whole tree?

And here are the template questions. When we write a function using the template, we fill in the template with the answers to these questions.

Let's see how the template questions help us define some functions that observe binary trees.

leaf-sum

What's the answer for a leaf?

```
leaf-sum : Tree -> Number
(define (leaf-sum t)
  (cond
    [(leaf? t) (leaf-datum t)]
    [else (+
           (leaf-sum (node-lson t))
           (leaf-sum (node-rson t)))]))
```

If you knew the answers for the 2 sons, how could you find the answer for the whole tree?

leaf-max

```
leaf-max : Tree -> Number
(define (leaf-max t)
  (cond
    [(leaf? t) (leaf-datum t)]
    [else (max
           (leaf-max (node-lson t))
           (leaf-max (node-rson t)))]))
```

What's the answer for a leaf?

If you knew the answers for the 2 sons, how could you find the answer for the whole tree?

leaf-min

```
leaf-min : Tree -> Number
(define (leaf-min t)
  (cond
    [(leaf? t) (leaf-datum t)]
    [else (min
           (leaf-min (node-lson t))
           (leaf-min (node-rson t)))]))
```

What's the answer for a leaf?

If you knew the answers for the 2 sons, how could you find the answer for the whole tree?

Summary

- You should now be able to:
  - Write a data definition for tree-structured information
  - Write a template for tree-structured information
  - Write functions that manipulate that data, using the template

Next Steps

- Study the file 05-1-trees.rkt in the Examples folder.
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 5.1
- Go on to the next lesson