Programming Languages and Compilers (CS 421)

Talia Ringer (they/them)

4218 SC, UIUC

https://courses.grainger.illinois.edu/cs421/fa2023/

Based heavily on slides by Elsa Gunter, which were based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Objectives for Today

- Last class, we took a step back, asking how would we actually **automate** transformations like CPS?
- We need needed a way to **represent** the syntax of our language that allows us to (1) **construct** a representation of a new (transformed) program, and (2) **match** over the syntax of the original
- We got that—**datatypes**
- Today, we'll continue covering **recursive datatypes**, including **mutually recursive datatypes**, emphasizing how they can represent the syntax of programs for transformations

Objectives for Today

- Last class, we took a step back, asking how would we actually **automate** transformations like CPS?
- We need needed a way to **represent** the syntax of our language that allows us to (1) **construct** a representation of a new (transformed) program, and (2) **match** over the syntax of the original
- We got that—**datatypes**
- Today, we'll continue covering **recursive datatypes**, including **mutually recursive datatypes**, emphasizing how these can represent the syntax of programs for transformations

Questions from Tuesday?

Recursive Datatypes, Continued

Reminder: Recursive Datatypes

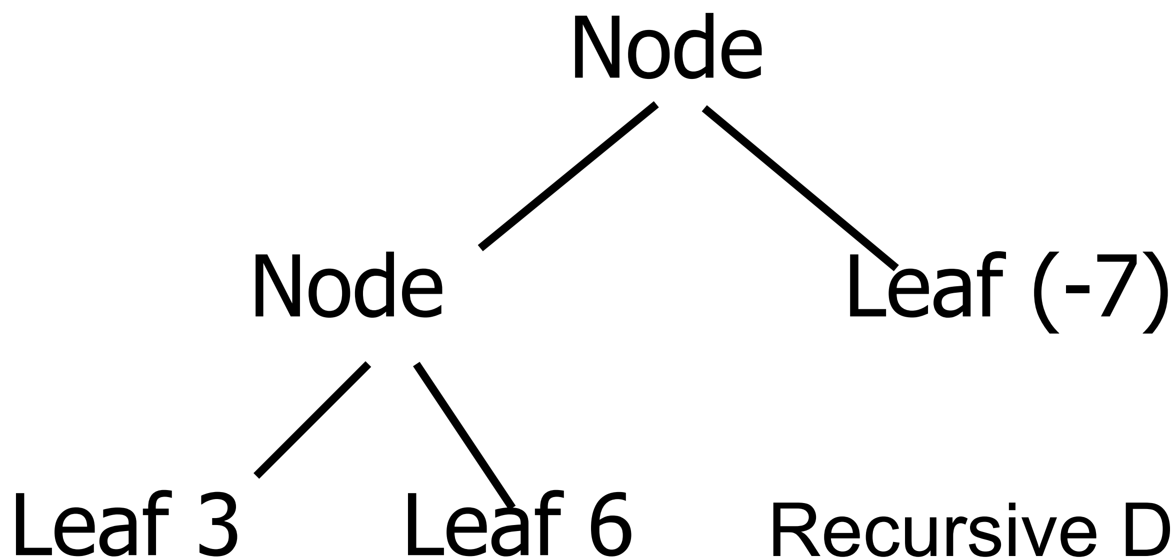
```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =
 Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```

Reminder: Recursive Datatypes

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =
 Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```

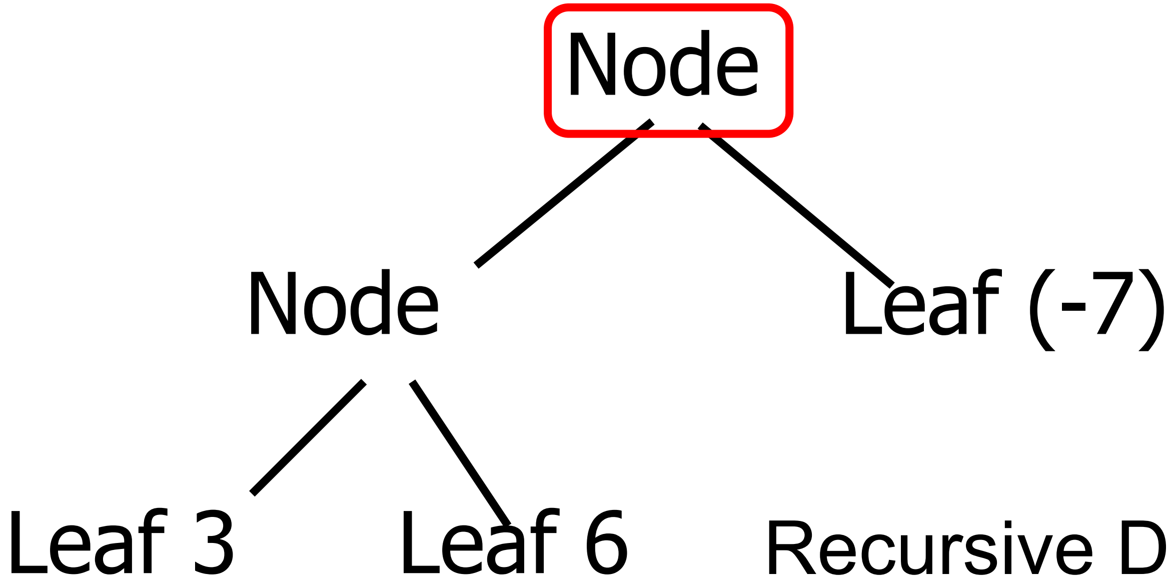
Reminder: Recursive Datatypes

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =
 Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```



Reminder: Recursive Datatypes

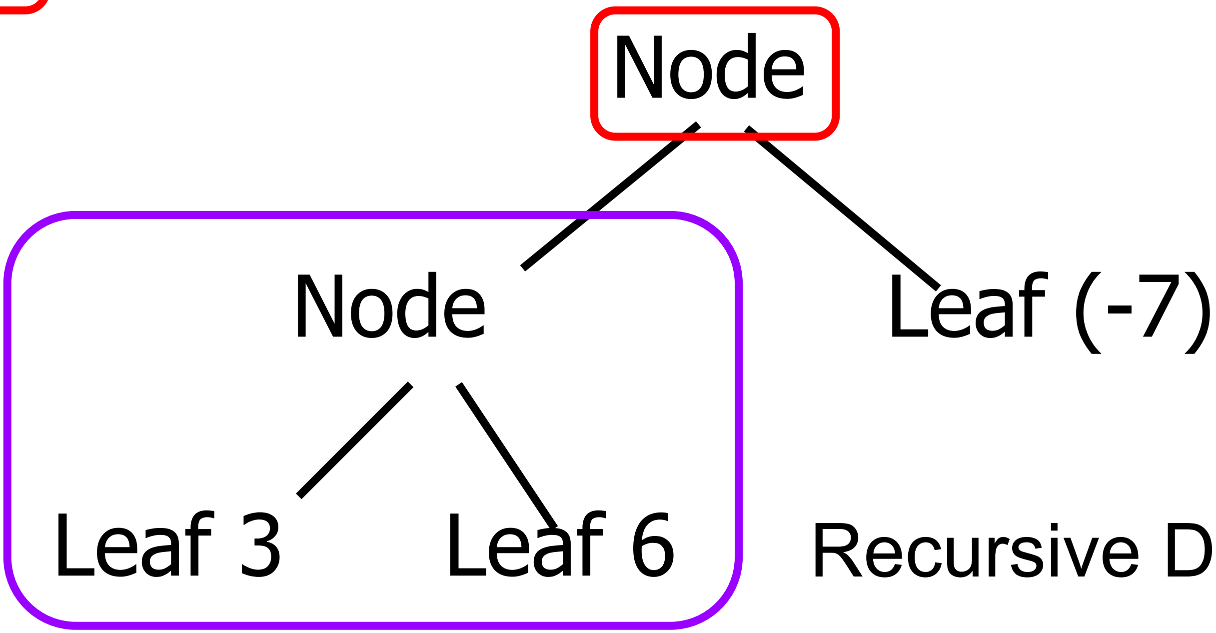
```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =
Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```



Reminder: Recursive Datatypes

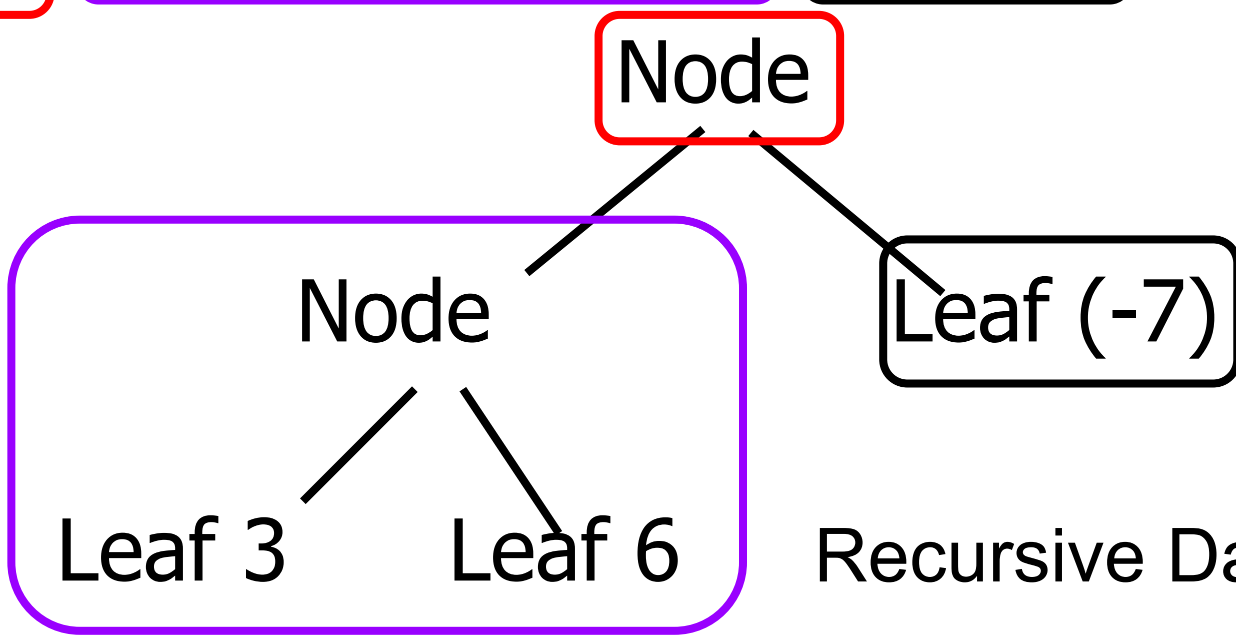
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =
Node (Node (Leaf 3, Leaf 6), Leaf (-7))



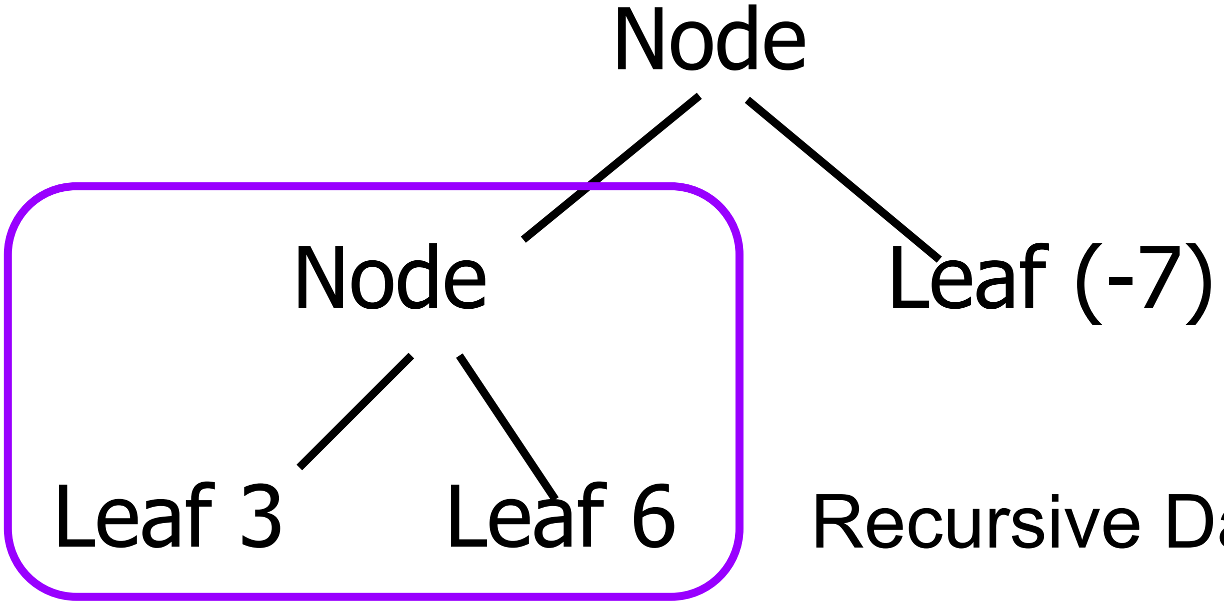
Reminder: Recursive Datatypes

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =
Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```



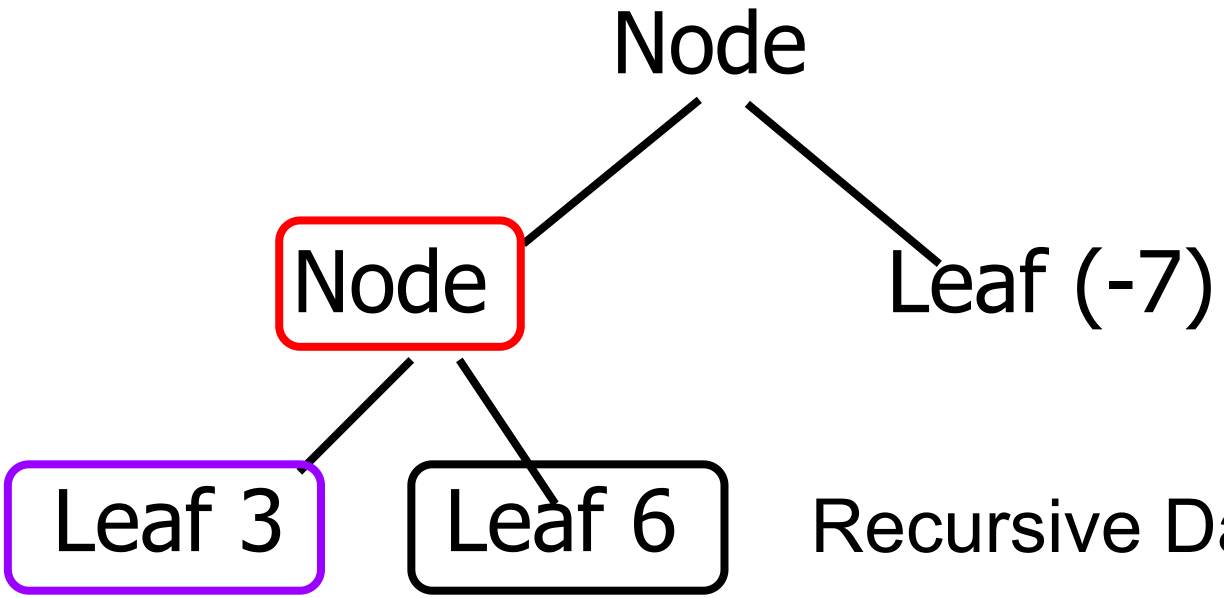
Reminder: Recursive Datatypes

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =
 Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```



Reminder: Recursive Datatypes

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =
 Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```



Recursive Functions

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

# let rec first_leaf_value tree =
    match tree with
    | (Leaf n) -> n
    | Node (l, r) -> first_leaf_value l;;
val first_leaf_value : int_Bin_Tree -> int = <fun>
# let left = first_leaf_value my_tree;;
val left : int = 3
```

Recursive Functions

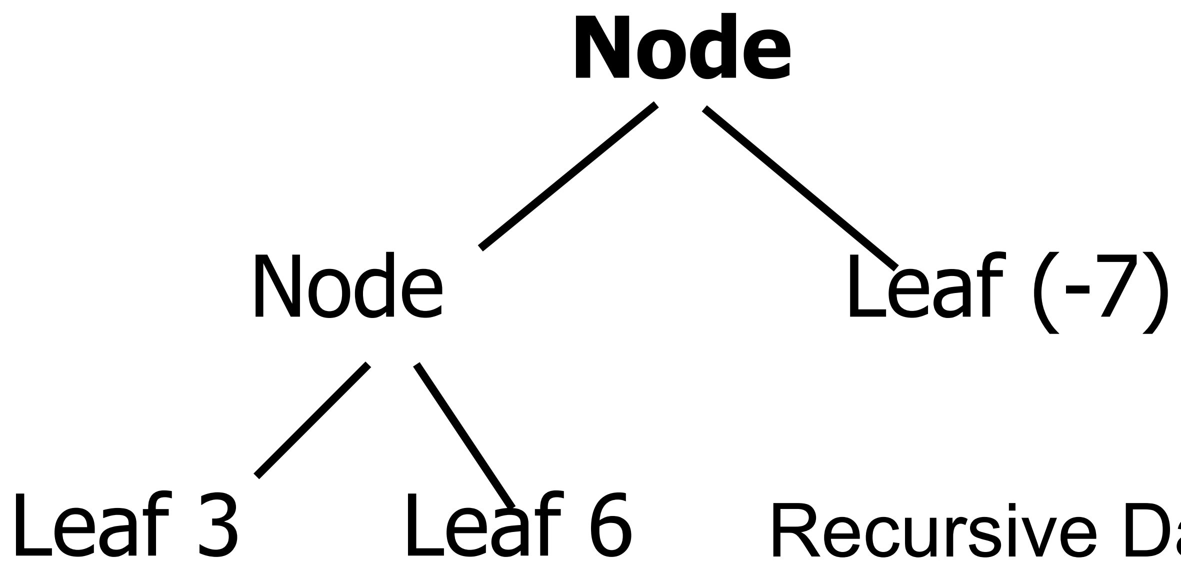
```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

# let rec first_leaf_value tree =
    match tree with
    | (Leaf n) -> n
    | Node (l, r) -> first_leaf_value l;;
val first_leaf_value : int_Bin_Tree -> int = <fun>
# let left = first_leaf_value my_tree;;
val left : int = 3
```

Recursive Functions

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

# let rec first_leaf_value tree =
    match tree with
    | (Leaf n) -> n
    | Node (l, r) -> first_leaf_value l;;
val first_leaf_value : int_Bin_Tree -> int = <fun>
# let left = first_leaf_value my_tree;;
val left : int = 3
```

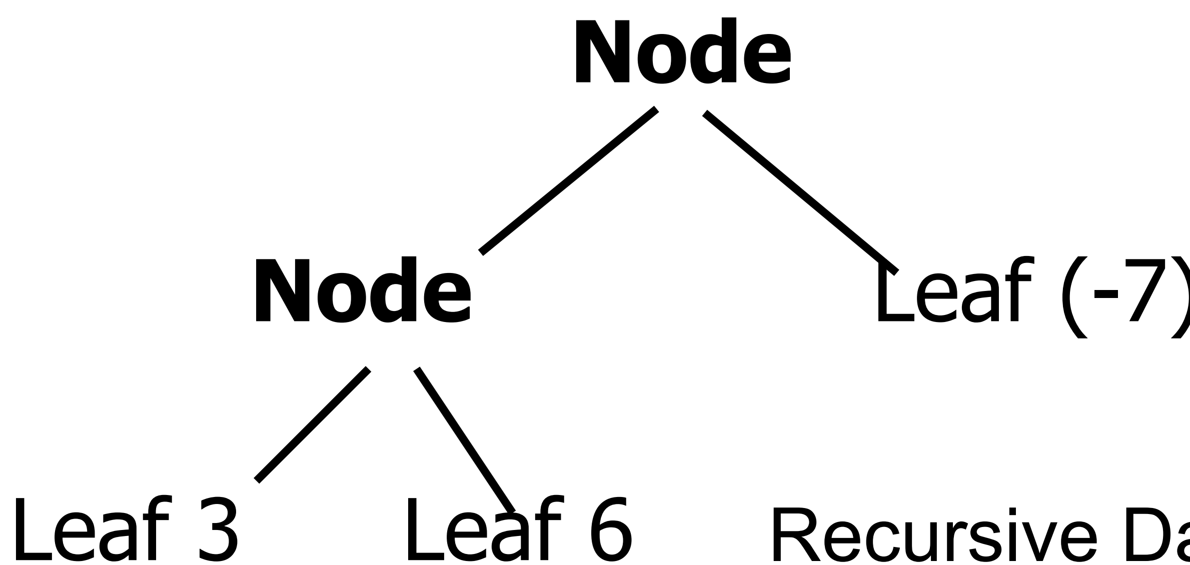
Recursive Functions

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =
 Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```



Recursive Functions

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =
 Node (Node (Leaf 3, Leaf 6), Leaf (-7))
```



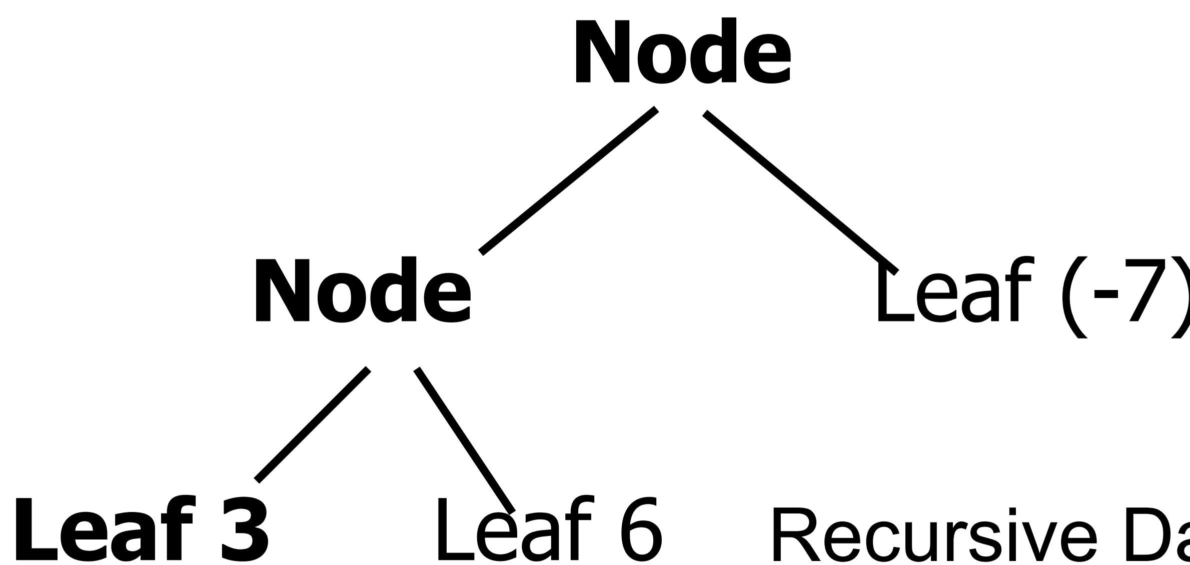
Recursive Functions

type int_Bin_Tree =

Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =

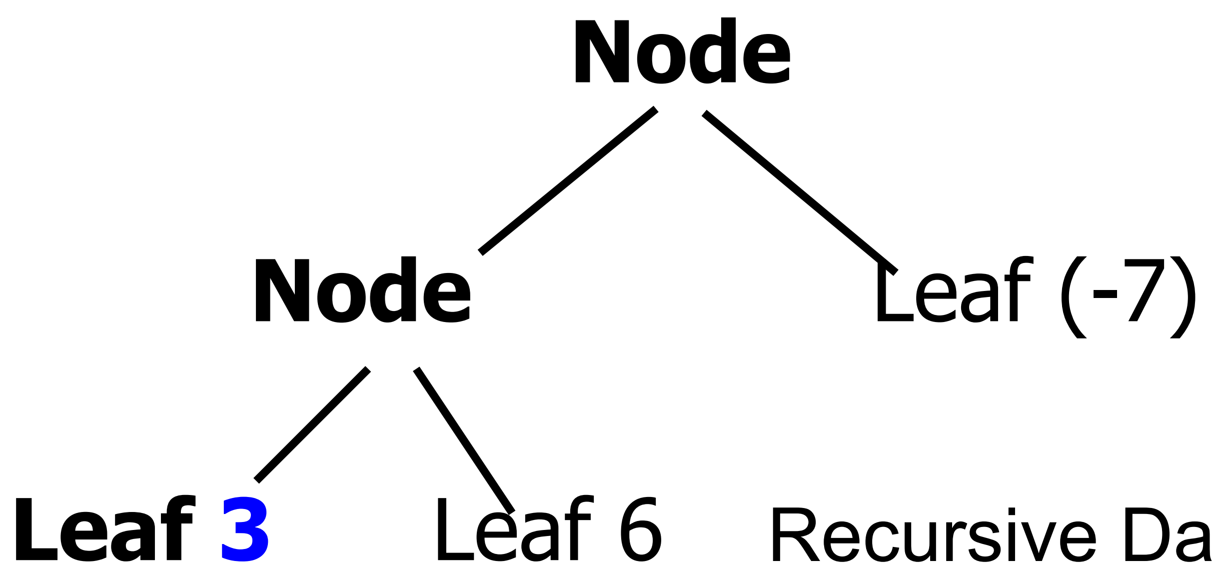
Node (Node (Leaf 3, Leaf 6), Leaf (-7))



Recursive Functions

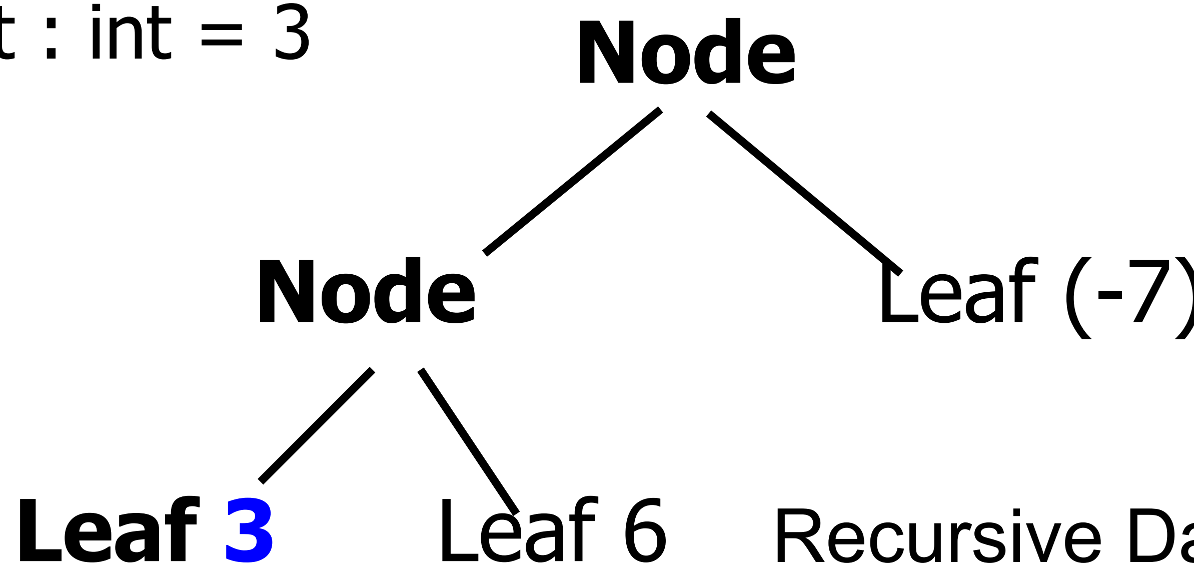
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

let my_tree =
 Node (Node (Leaf 3, Leaf 6), Leaf (-7))



Recursive Functions

```
# let rec first_leaf_value tree =
    match tree with
    | (Leaf n) -> n
    | Node (l, r) -> first_leaf_value l;;
# let left = first_leaf_value my_tree;;
val left : int = 3
```



Problem

type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

Write sum_tree : int_Bin_Tree -> int

(* adds all ints in an int_Bin_Tree *)
let rec sum_tree t =

Problem

type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

Write sum_tree : int_Bin_Tree -> int

(* adds all ints in an int_Bin_Tree *)
let rec sum_tree t =

What’s the first thing we do?

Problem

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)
```

Write sum_tree : int_Bin_Tree -> int

```
(* adds all ints in an int_Bin_Tree *)
let rec sum_tree t =
  match t with
```

What are the cases?

Problem

type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)

Write sum_tree : int_Bin_Tree -> int

```
(* adds all ints in an int_Bin_Tree *)
let rec sum_tree t =
  match t with
  | Leaf n ->
  | Node (l, r) ->
```

Problem

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)
```

Write sum_tree : int_Bin_Tree -> int

```
(* adds all ints in an int_Bin_Tree *)
let rec sum_tree t =
  match t with
  | Leaf n ->
  | Node (l, r) ->
```

What's the base case?

Problem

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)
```

Write sum_tree : int_Bin_Tree -> int

```
(* adds all ints in an int_Bin_Tree *)
let rec sum_tree t =
  match t with
  | Leaf n -> n
  | Node (l, r) ->
```

Problem

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)
```

Write sum_tree : int_Bin_Tree -> int

```
(* adds all ints in an int_Bin_Tree *)
let rec sum_tree t =
  match t with
  | Leaf n -> n
  | Node (l, r) ->
```

What's the recursive case?

Problem

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)
```

Write sum_tree : int_Bin_Tree -> int

```
(* adds all ints in an int_Bin_Tree *)
let rec sum_tree t =
  match t with
  | Leaf n -> n
  | Node (l, r) -> sum_tree l + sum_tree r
```

Problem

```
type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree)
```

```
(* adds all ints in an int_Bin_Tree *)
let rec sum_tree t =
  match t with
  | Leaf n -> n
  | Node (l, r) -> sum_tree l + sum_tree r
```

Questions so far?

Representing Language Syntax

Recursive Data Types in Languages!

```
# type mon_op = ...
# type bin_op = IntPlusOp |  IntMinusOp |  EqOp | ...
# type const = BoolConst of bool | IntConst of int | ...
# type exp =
  | VarExp of string (* variables *)
  | ConstExp of const (* constants *)
  | MonOpAppExp of mon_op * exp (* unary ops *)
  | BinOpAppExp of bin_op * exp * exp (* bin ops *)
  | IfExp of exp * exp * exp (* conditionals *)
  | AppExp of exp * exp (* function application *)
  | FunExp of string * exp (* functions *)
```

Recursive Data Types in Languages!

```
# type mon_op = ...
# type bin_op = IntPlusOp | IntMinusOp | EqOp | ...
# type const = BoolConst of bool | IntConst of int | ...
# type exp =
   | VarExp of string (* variables *)
   | ConstExp of const (* constants *)
   | MonOpAppExp of mon_op * exp (* unary ops *)
   | BinOpAppExp of bin_op * exp * exp (* bin ops *)
   | IfExp of exp * exp * exp (* conditionals *)
   | AppExp of exp * exp (* function application *)
   | FunExp of string * exp  (* functions *)
```

Recursive Data Types in Languages!

```
# type mon_op = ...
# type bin_op = IntPlusOp |  IntMinusOp |  EqOp | ...
# type const = BoolConst of bool | IntConst of int | ...
# type exp =
   | VarExp of string (* variables *)
   | ConstExp of const (* constants *)
   | MonOpAppExp of mon_op * exp (* unary ops *)
   | BinOpAppExp of bin_op * exp * exp (* bin ops *)
   | IfExp of exp * exp * exp (* conditionals *)
   | AppExp of exp * exp (* function application *)
   | FunExp of string * exp  (* functions *)
```

Recursive Data Types in Languages!

```
# type mon_op = …
# type bin_op = IntPlusOp |  IntMinusOp |  EqOp | …
# type const = BoolConst of bool | IntConst of int | …
# type exp =
   | VarExp of string (* variables *)
   | ConstExp of const (* constants *)
   | MonOpAppExp of mon_op * exp (* unary ops *)
   | BinOpAppExp of bin_op * exp * exp (* bin ops *)
   | IfExp of exp * exp * exp (* conditionals *)
   | AppExp of exp * exp (* function application *)
   | FunExp of string * exp  (* functions *)
```

Recursive Data Types in Languages!

```
# type mon_op = ...
# type bin_op = IntPlusOp |  IntMinusOp |  EqOp | ...
# type const = BoolConst of bool | IntConst of int | ...
# type exp =
   | VarExp of string
   | ConstExp of const
   | MonOpAppExp of mon_op * exp
   | BinOpAppExp of bin_op * exp * exp
   | IfExp of exp * exp * exp
   | AppExp of exp * exp
   | FunExp of string * exp
```

How to represent 6?

Recursive Data Types in Languages!

```
# type mon_op = ...
# type bin_op = IntPlusOp |  IntMinusOp |  EqOp | ...
# type const = BoolConst of bool | IntConst of int | ...
# type exp =
   | VarExp of string
   | ConstExp of const
   | MonOpAppExp of mon_op * exp
   | BinOpAppExp of bin_op * exp * exp
   | IfExp of exp * exp * exp
   | AppExp of exp * exp
   | FunExp of string * exp
```

How to represent 6?

??? (IntConst 6)

Recursive Data Types in Languages!

type mon_op = …
type bin_op = IntPlusOp | IntMinusOp | EqOp | …
type const = BoolConst of bool | **IntConst** of **int** | …
type exp =
 | VarExp of string
 | **ConstExp** of **const**
 | MonOpAppExp of mon_op * exp
 | BinOpAppExp of bin_op * exp * exp
 | IfExp of exp * exp * exp
 | AppExp of exp * exp
 | FunExp of string * exp

How to represent 6?

ConstExp (IntConst 6)

Recursive Data Types in Languages!

- # type mon_op = ...
- # type bin_op = IntPlusOp | IntMinusOp | EqOp | ...
- # type const = BoolConst of bool | **IntConst** of **int** | ...
- # type exp =
  - | VarExp of string
  - | **ConstExp** of **const**
  - | MonOpAppExp of mo
  - | BinOpAppExp of bin_
  - | IfExp of exp * exp * exp
  - | AppExp of exp * exp
  - | FunExp of string * exp

How to represent 6 + 5?

???
(ConstExp (IntConst 6))
(ConstExp (IntConst 5))

Recursive Data Types in Languages!

```
# type mon_op = ...
# type bin_op = IntPlusOp |  IntMinusOp |  EqOp | ...
# type const = BoolConst of bool | IntConst of int | ...
# type exp =
    | VarExp of string
    | ConstExp of const
    | MonOpAppExp of mo
    | BinOpAppExp of bin_
    | IfExp of exp * exp * exp
    | AppExp of exp * exp
    | FunExp of string * exp
```

How to represent 6 + 5?

```
??? (IntPlusOp
  (ConstExp (IntConst 6))
  (ConstExp (IntConst 5)))
```

Recursive Data Types in Languages!

```
# type mon_op = ...
# type bin_op = IntPlusOp |  IntMinusOp |  EqOp | ...
# type const = BoolConst of bool | IntConst of int | ...
# type exp =
    | VarExp of string
    | ConstExp of const
    | MonOpAppExp of mo
    | BinOpAppExp of b
    | IfExp of exp * exp * exp
    | AppExp of exp * exp
    | FunExp of string * exp
```

How to represent 6 + 5?

BinOppAppExp (IntPlusOp (ConstExp (IntConst 6)) (ConstExp (IntConst 5)))

✓

Representing Language Syntax

?

Representing Language Semantics?

Could Swap If/Else Cases…

```
# type mon_op = …
# type bin_op = IntPlusOp |  IntMinusOp |  EqOp | …
# type const = BoolConst of bool | IntConst of int | …
# type exp =
   | VarExp of string
   | ConstExp of const
   | MonOpAppExp of mon_op * exp
   | BinOpAppExp of bin_op * exp * exp
   | IfExp of exp * exp * exp
   | AppExp of exp * exp
   | FunExp of string * exp
```

?

Representing Language Semantics?

Questions so far?

Functions About Language Syntax

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
  | BinOpAppExp of bin_op * exp * exp
  | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
    | BinOpAppExp of bin_op * exp * exp
    | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

```
# let rec varCnt exp =
    match exp with
    | VarExp x ->
    | ConstExp c ->
    | BinOpAppExp (b, e1, e2) ->
    | FunExp (x, e) ->
    | AppExp (e1, e2) ->
```

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
   | BinOpAppExp of bin_op * exp * exp
   | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

```
# let rec varCnt exp =
   match exp with
   | VarExp x ->
   | ConstExp c ->
   | BinOpAppExp (b, e1, e2) ->
   | FunExp (x, e) ->
   | AppExp (e1, e2) ->
```

How many variables are in a variable?

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
   | BinOpAppExp of bin_op * exp * exp
   | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

```
# let rec varCnt exp =
   match exp with
   | VarExp x -> 1
   | ConstExp c ->
   | BinOpAppExp (b, e1, e2) ->
   | FunExp (x, e) ->
   | AppExp (e1, e2) ->
```

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
    | BinOpAppExp of bin_op * exp * exp
    | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

```
# let rec varCnt exp =
    match exp with
    | VarExp x -> 1
    | ConstExp c ->
    | BinOpAppExp (b, e1, e2) ->
    | FunExp (x, e) ->
    | AppExp (e1, e2) ->
```

How many variables are in a constant?

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
    | BinOpAppExp of bin_op * exp * exp
    | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

```
# let rec varCnt exp =
   match exp with
    | VarExp x -> 1
    | ConstExp c -> 0
    | BinOpAppExp (b, e1, e2) ->
    | FunExp (x, e) ->
    | AppExp (e1, e2) ->
```

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
    | BinOpAppExp of bin_op * exp * exp
    | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

```
# let rec varCnt exp =
    match exp with
    | VarExp x -> 1
    | ConstExp c -> 0
    | BinOpAppExp (b, e1, e2) ->
    | FunExp (x, e) ->
    | AppExp (e1, e2) ->
```

How many variables are in (b e1 e2)?

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
   | BinOpAppExp of bin_op * exp * exp
   | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

```
# let rec varCnt exp =
   match exp with
   | VarExp x -> 1
   | ConstExp c -> 0
   | BinOpAppExp (b, e1, e2) -> varCnt e1 + varCnt e2
   | FunExp (x, e) ->
   | AppExp (e1, e2) ->
```

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
  | BinOpAppExp of bin_op * exp * exp
  | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

```
# let rec varCnt exp =
  match exp with
  | VarExp x -> 1
  | ConstExp c -> 0
  | BinOpAppExp (b, e1, e2) -> varCnt e1 + varCnt e2
  | FunExp (x, e) ->
  | AppExp (e1, e2) -> varCnt e1 + varCnt e2
```

(The app case is similar.)

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
    | BinOpAppExp of bin_op * exp * exp
    | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

```
# let rec varCnt exp =
    match exp with
    | VarExp x -> 1
    | ConstExp c -> 0
    | BinOpAppExp (b, e1, e2) -> varCnt e1 + varCnt e2
    | FunExp (x, e) ->
    | AppExp (e1, e2) -> varCnt e1 + varCnt e2
```

How many variables in a function from arg x to body e? Depends …

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
    | BinOpAppExp of bin_op * exp * exp
    | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

```
# let rec varCnt exp =
   match exp with
    | VarExp x -> 1
    | ConstExp c -> 0
    | BinOpAppExp (b, e1, e2) -> varCnt e1 + varCnt e2
    | FunExp (x, e) -> varCnt e
    | AppExp (e1, e2) -> varCnt e1 + varCnt e2
```

How many variables in a function from arg x to body e, not counting x?

Recursion over Recursive Data Types

```
# type exp = VarExp of string | ConstExp of const
   | BinOpAppExp of bin_op * exp * exp
   | FunExp of string * exp | AppExp of exp * exp
```

- How to count the number of variables in an exp?

```
# let rec varCnt exp =
   match exp with
   | VarExp x -> 1
   | ConstExp c -> 0
   | BinOpAppExp (b, e1, e2) -> varCnt e1 + varCnt e2
   | FunExp (x, e) -> 1 + varCnt e
   | AppExp (e1, e2) -> varCnt e1 + varCnt e2
```

How many variables in a function from arg x to body e, counting x?

✓

Reasoning About Syntax

✓

Representing Language Semantics?

Questions so far?

Your turn: Try Problem 3 on MP5

Mapping Over Recursive Types

Mapping over Recursive Types

```
# let rec ibtreeMap f tree =
    match tree with
    | Leaf n -> Leaf (f n)
    | Node (l, r) -> Node (ibtreeMap f l, ibtreeMap f r);;
val ibtreeMap :
  (int -> int) -> int_Bin_Tree -> int_Bin_Tree = <fun>
```

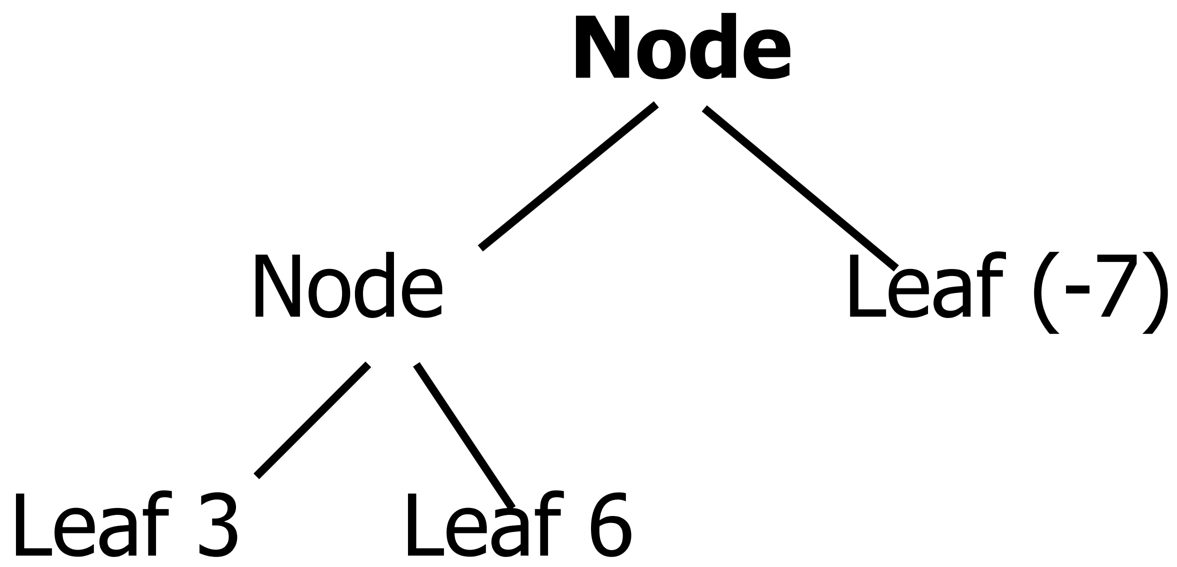
Mapping over Recursive Types

```
# let rec ibtreeMap f tree =
    match tree with
    | Leaf n -> Leaf (f n)
    | Node (l, r) -> Node (ibtreeMap f l, ibtreeMap f r);;
val ibtreeMap :
  (int -> int) -> int_Bin_Tree -> int_Bin_Tree = <fun>
```

Map

Mapping over Recursive Types

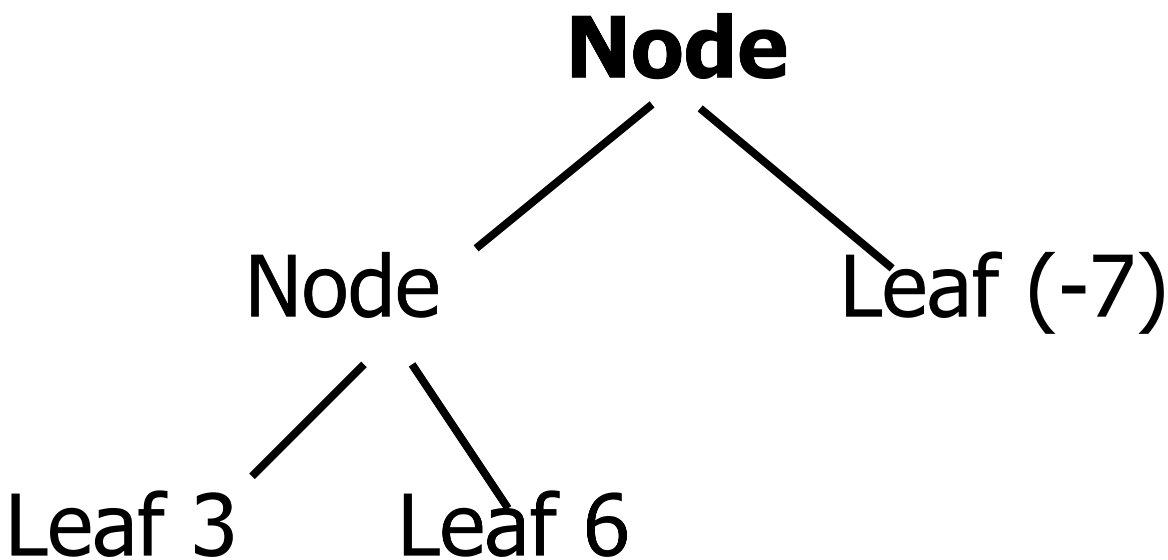
```
# let rec ibtreeMap f tree =
    match tree with
    | Leaf n -> Leaf (f n)
    | Node (l, r) -> Node (ibtreeMap f l, ibtreeMap f r);;
```



Map

Mapping over Recursive Types

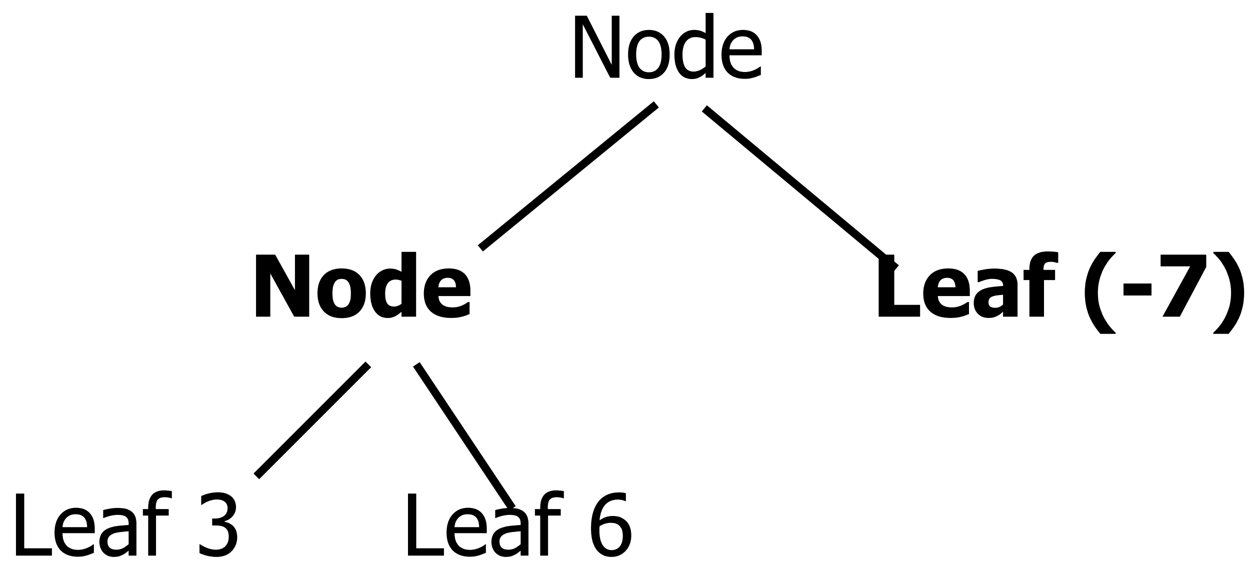
```
# let rec ibtreeMap f tree =
    match tree with
    | Leaf n -> Leaf (f n)
    | Node (l, r) -> Node (ibtreeMap f l, ibtreeMap f r);;
```



Map

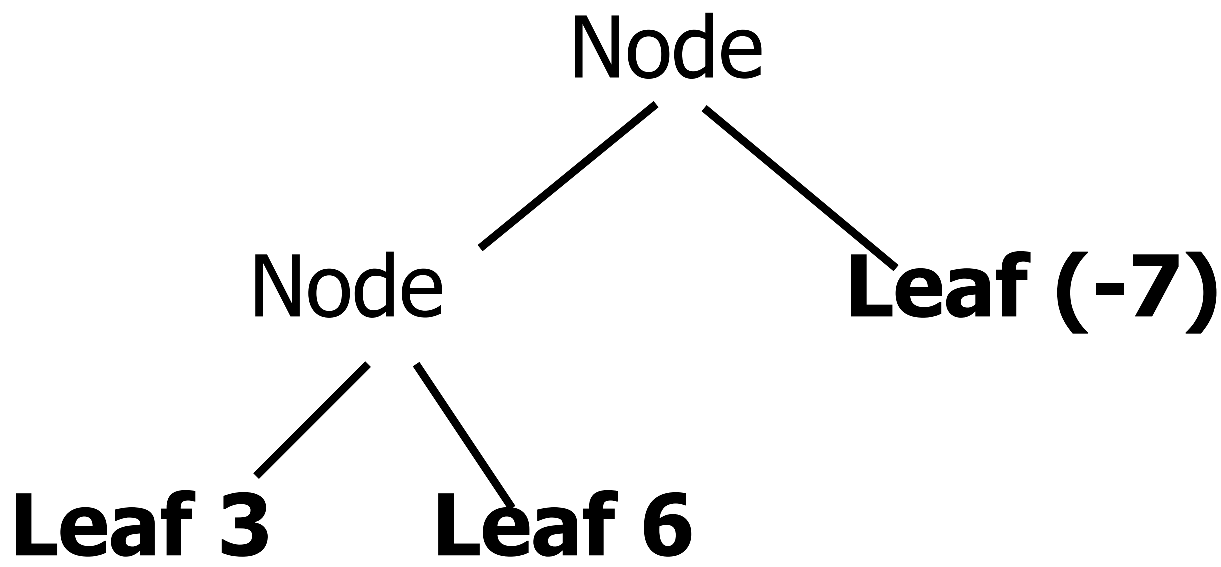
Mapping over Recursive Types

```
# let rec ibtreeMap f tree =
    match tree with
    | Leaf n -> Leaf (f n)
    | Node (l, r) -> Node (ibtreeMap f l, ibtreeMap f r);;
```



Mapping over Recursive Types

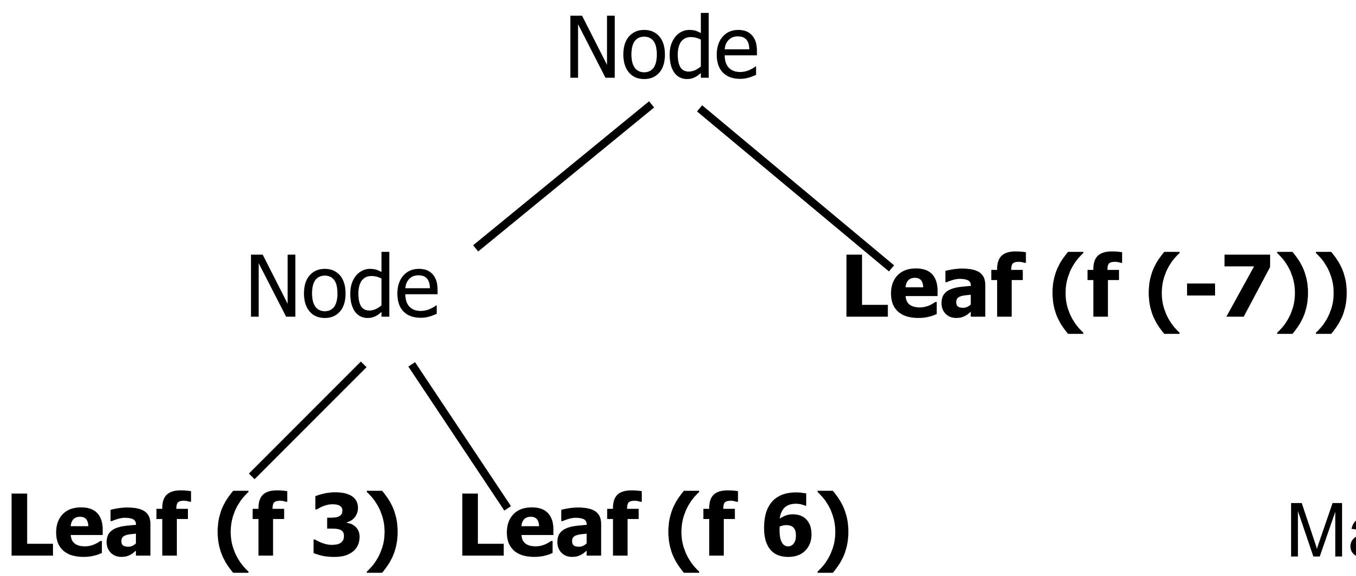
```
# let rec ibtreeMap f tree =
    match tree with
    | Leaf n -> Leaf (f n)
    | Node (l, r) -> Node (ibtreeMap f l, ibtreeMap f r);;
```



Map

Mapping over Recursive Types

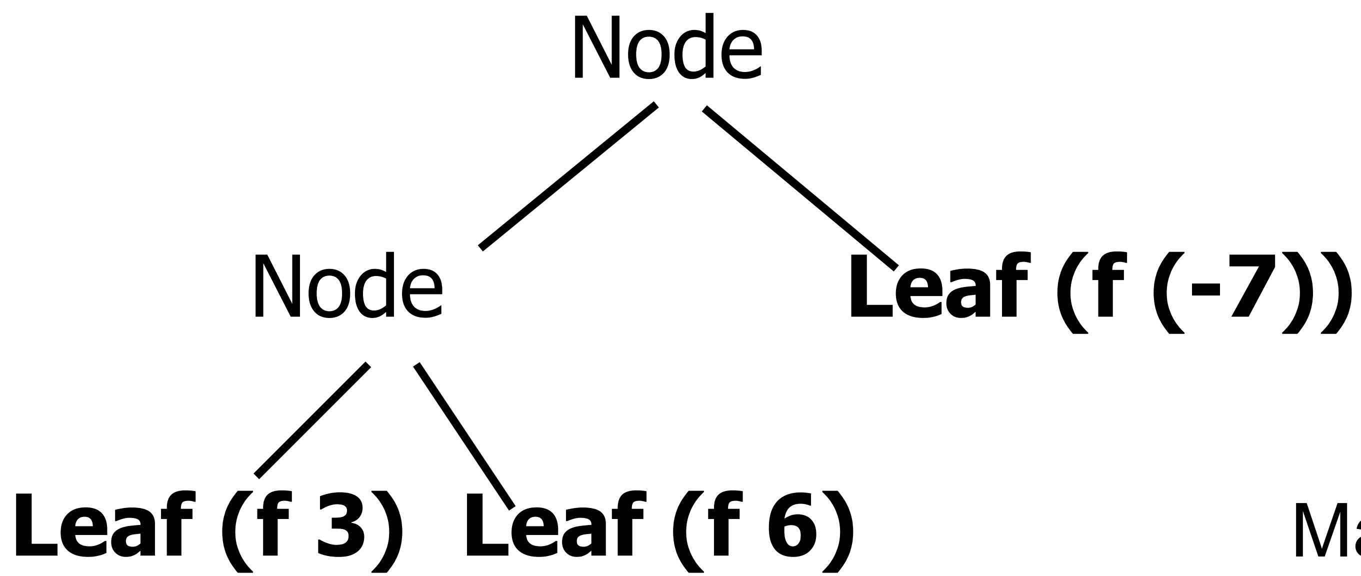
```
# let rec ibtreeMap f tree =
    match tree with
    | Leaf n -> Leaf (f n)
    | Node (l, r) -> Node (ibtreeMap f l, ibtreeMap f r);;
```



Map

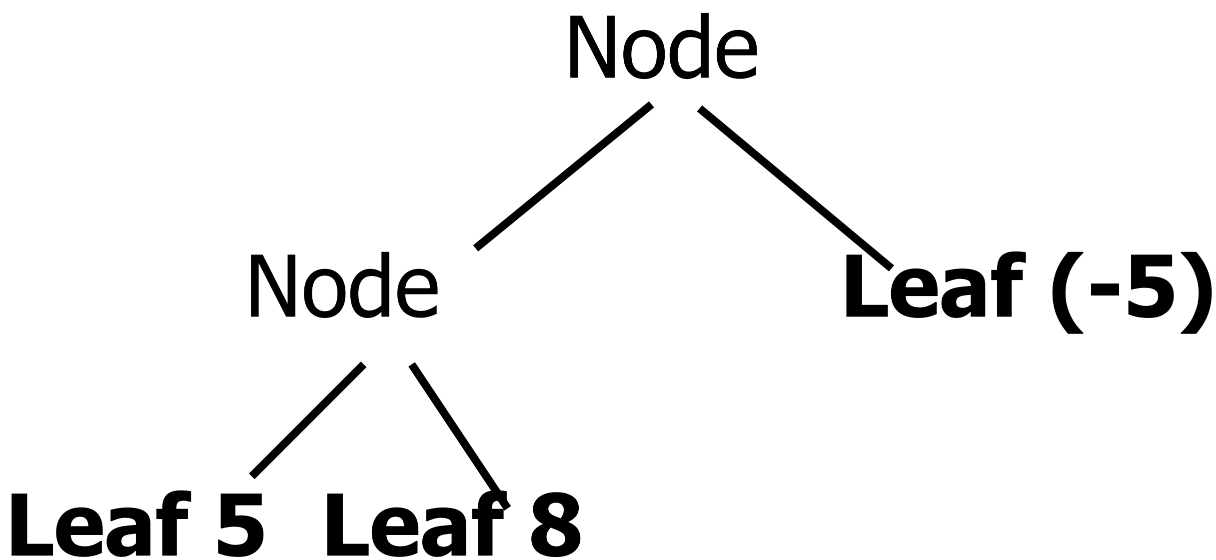
Mapping over Recursive Types

```
# let rec ibtreeMap f tree =
    match tree with
    | Leaf n -> Leaf (f n)
    | Node (l, r) -> Node (ibtreeMap f l, ibtreeMap f r);;
# ibtreeMap ((+) 2) my_tree;;
```



Mapping over Recursive Types

```
# let rec ibtreeMap f tree =
    match tree with
    | Leaf n -> Leaf (f n)
    | Node (l, r) -> Node (ibtreeMap f l, ibtreeMap f r);;
# ibtreeMap ((+) 2) my_tree;;
```



Mapping over Recursive Types

```
# let rec ibtreeMap f tree =
    match tree with
    | Leaf n -> Leaf (f n)
    | Node (l, r) -> Node (ibtreeMap f l, ibtreeMap f r);;
# ibtreeMap ((+) 2) my_tree;;
- : int_Bin_Tree =
  Node (Node (Leaf 5, Leaf 8), Leaf (-5))
```

Map

Folding Over Recursive Types

Folding Over Recursive Types

Caveat: “left” and “right” no longer make sense in general. One canonical fold; others are quirks of symmetry as with lists.

Folding Over Recursive Types

Caveat: Folks tend to call the general fold a "right" fold though.

Folding over Recursive Types

```
# let rec ibtreeFoldRight leafFun nodeFun tree =
   match tree with
   | Leaf n -> leafFun n
   | Node (l, r) ->
       nodeFun
         (ibtreeFoldRight leafFun nodeFun l)
         (ibtreeFoldRight leafFun nodeFun r);;
val ibtreeFoldRight :
  (int -> 'a) -> ('a -> 'a -> 'a) -> int_Bin_Tree -> 'a
= <fun>
```

Folding over Recursive Types

```
# let rec ibtreeFoldRight leafFun nodeFun tree =
    match tree with
    | Leaf n -> leafFun n
    | Node (l, r) ->
        nodeFun
          (ibtreeFoldRight leafFun nodeFun l)
          (ibtreeFoldRight leafFun nodeFun r);;
val ibtreeFoldRight :
  (int -> 'a) -> ('a -> 'a -> 'a) -> int_Bin_Tree -> 'a
= <fun>
```

How to transform data?

Folding over Recursive Types

```
# let rec ibtreeFoldRight leafFun nodeFun tree =
    match tree with
    | Leaf n -> leafFun n
    | Node (l, r) ->
        nodeFun
          (ibtreeFoldRight leafFun nodeFun l)
          (ibtreeFoldRight leafFun nodeFun r);;
val ibtreeFoldRight :
  (int -> 'a) -> ('a -> 'a -> 'a) -> int_Bin_Tree -> 'a
= <fun>
```

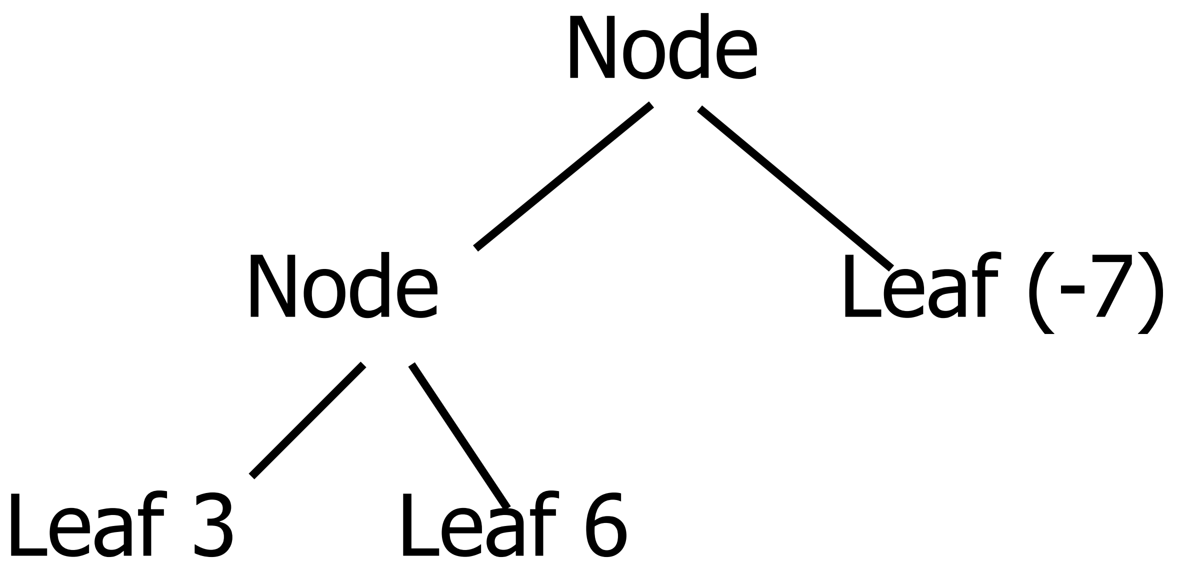
How to combine subtree results?

Folding over Recursive Types

```
# let tree_sum = ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum my_tree;;
- : int = 2
```

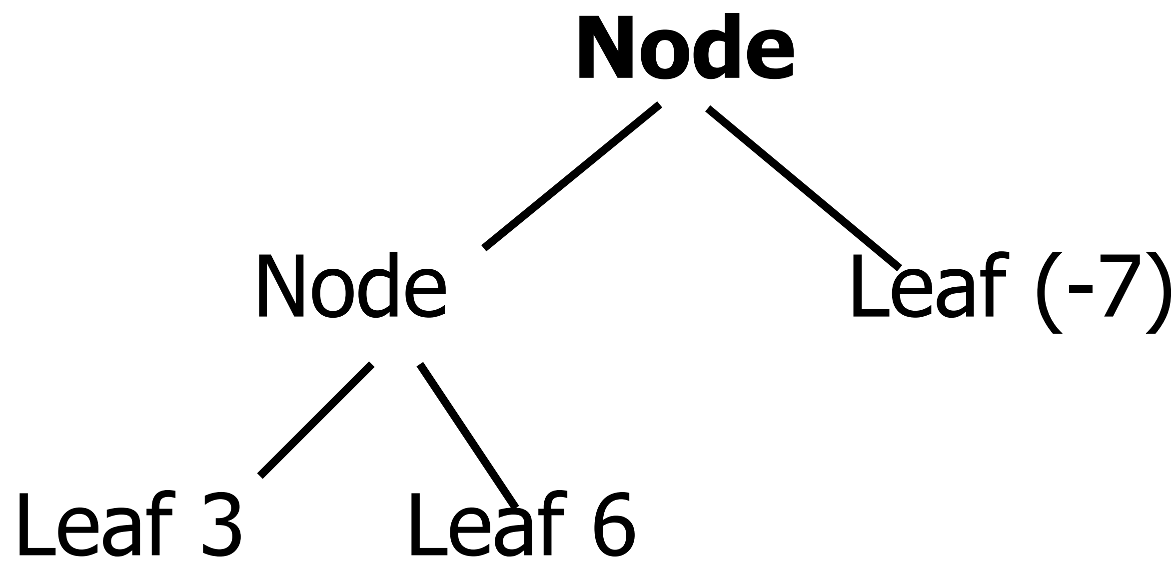
Folding over Recursive Types

```
# let tree_sum = ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum my_tree;;
- : int = 2
```



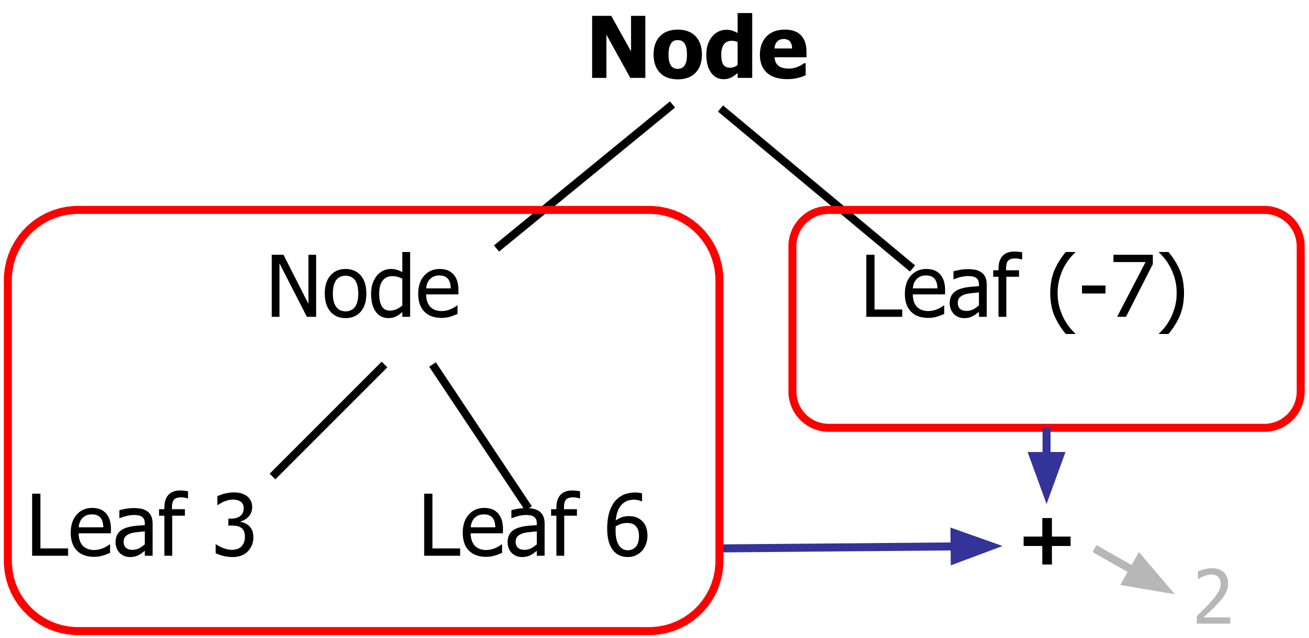
Folding over Recursive Types

```
# let tree_sum = ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum my_tree;;
- : int = 2
```



Folding over Recursive Types

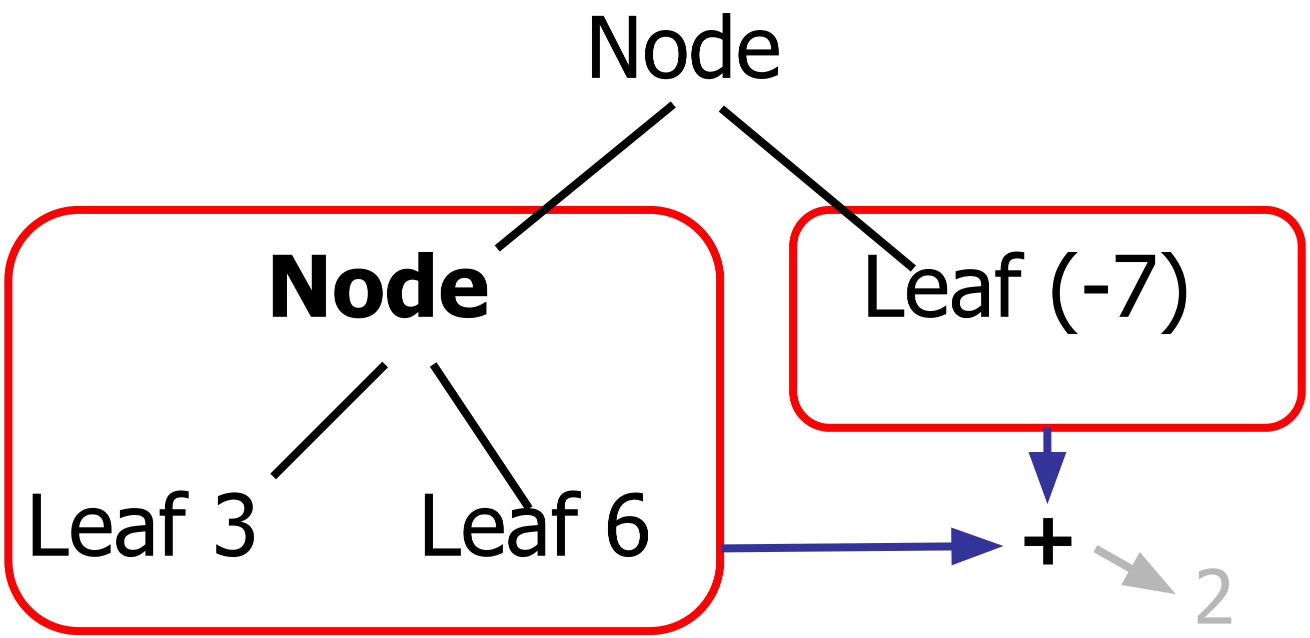
```
# let tree_sum = ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum my_tree;;
- : int = 2
```



Fold

Folding over Recursive Types

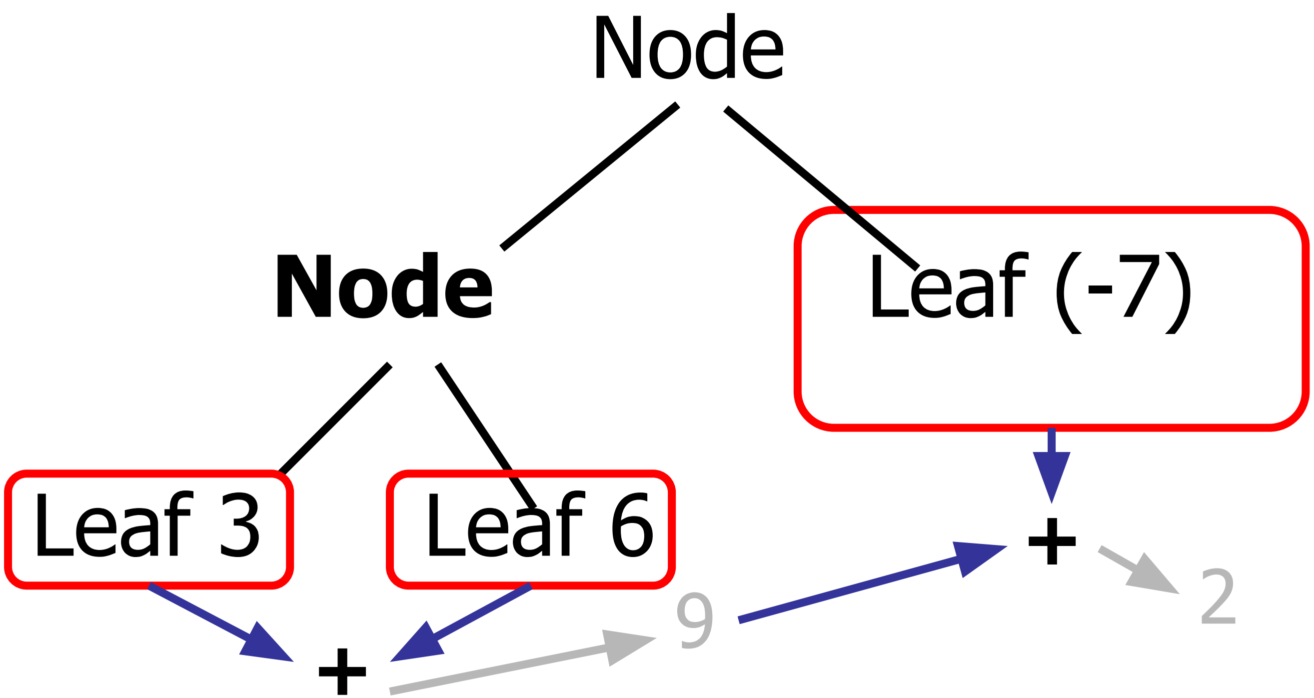
```
# let tree_sum = ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum my_tree;;
- : int = 2
```



Fold

Folding over Recursive Types

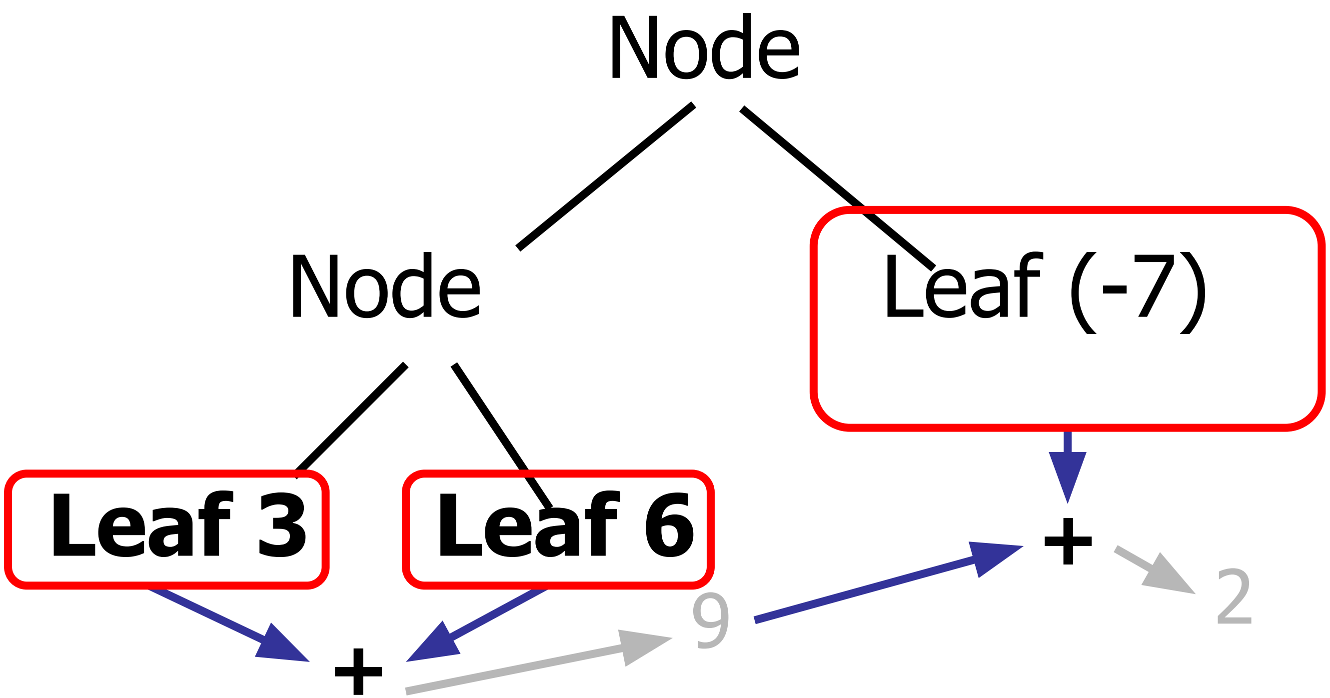
```
# let tree_sum = ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum my_tree;;
- : int = 2
```



Fold

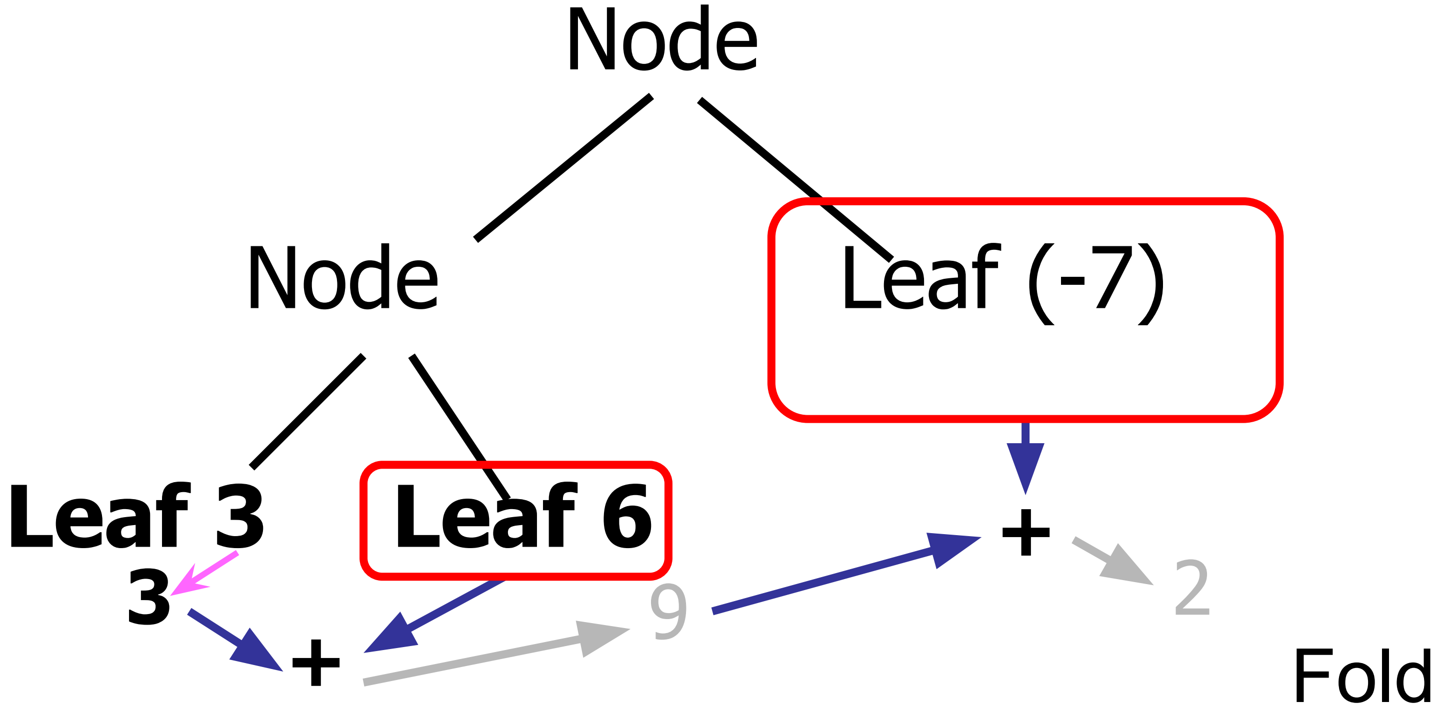
Folding over Recursive Types

```
# let tree_sum = ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum my_tree;;
- : int = 2
```



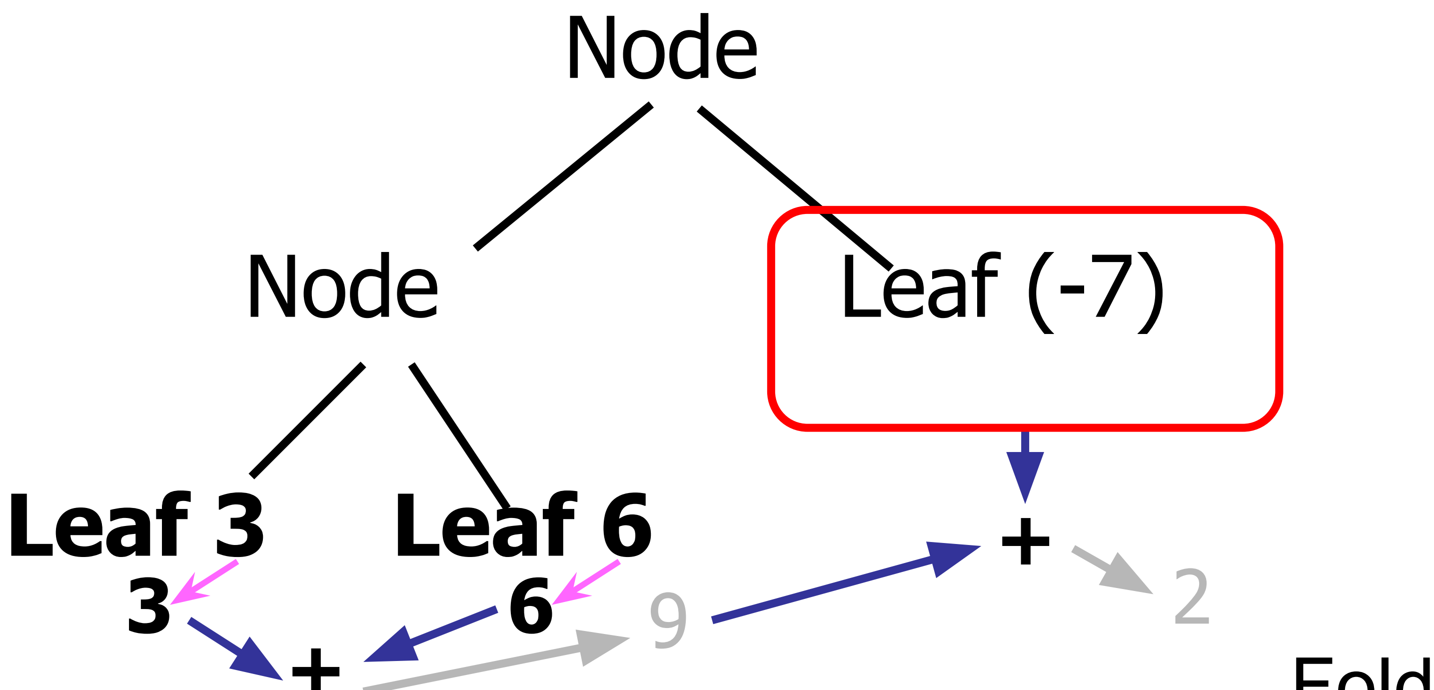
Folding over Recursive Types

```
# let tree_sum = ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum my_tree;;
- : int = 2
```



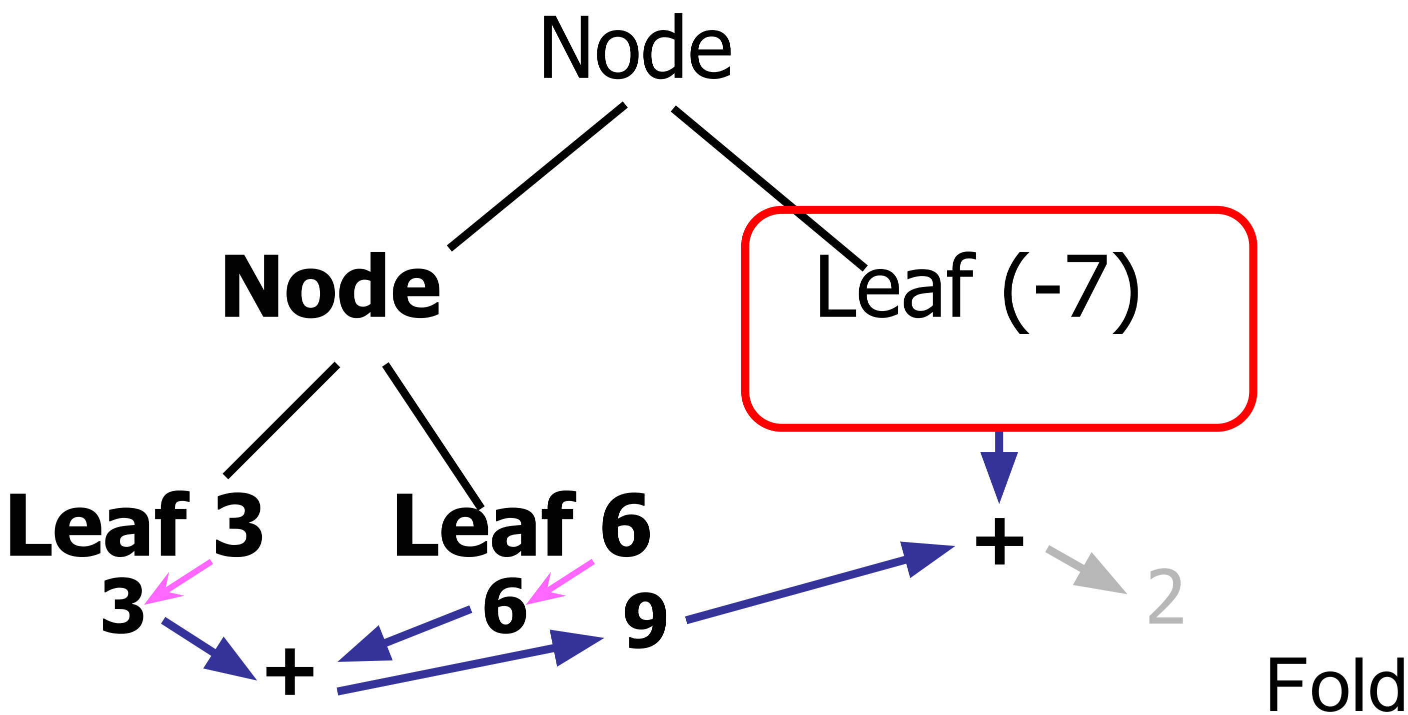
Folding over Recursive Types

```
# let tree_sum = ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum my_tree;;
- : int = 2
```



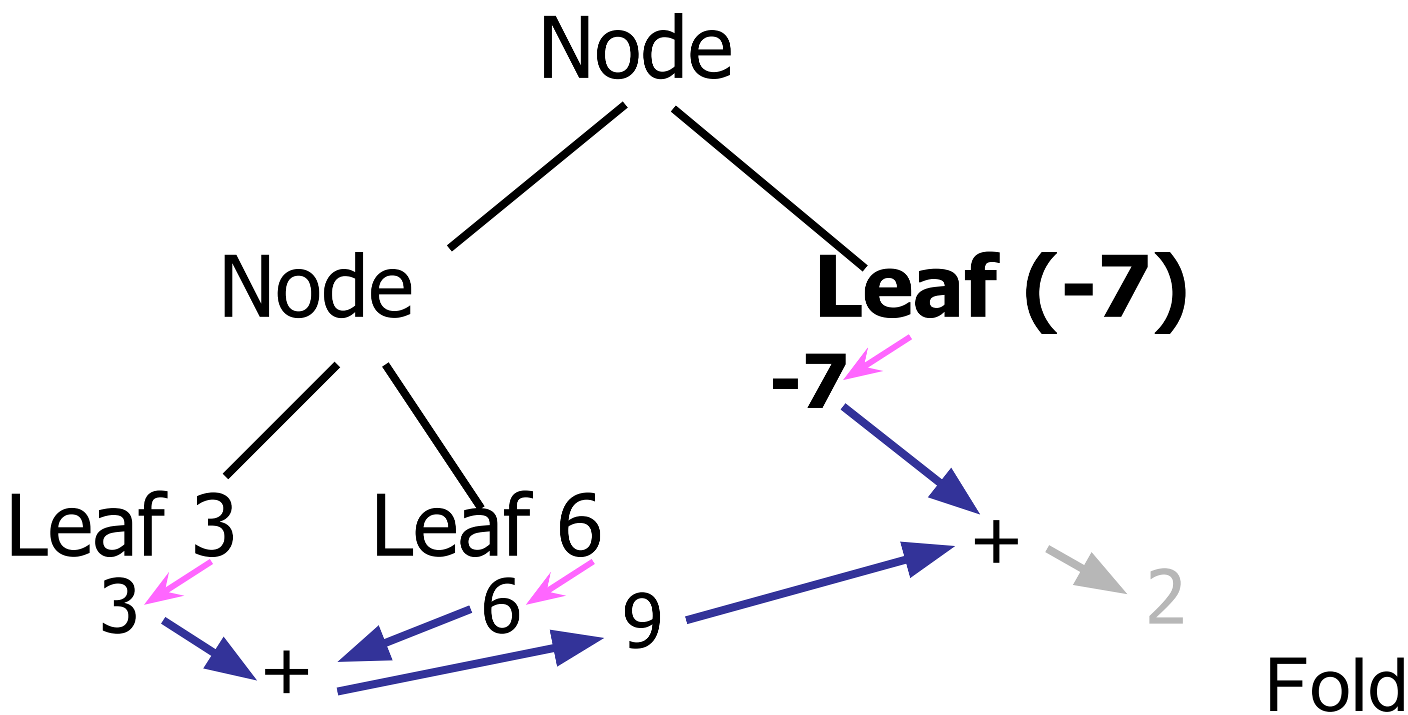
Folding over Recursive Types

```
# let tree_sum = ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum my_tree;;
- : int = 2
```



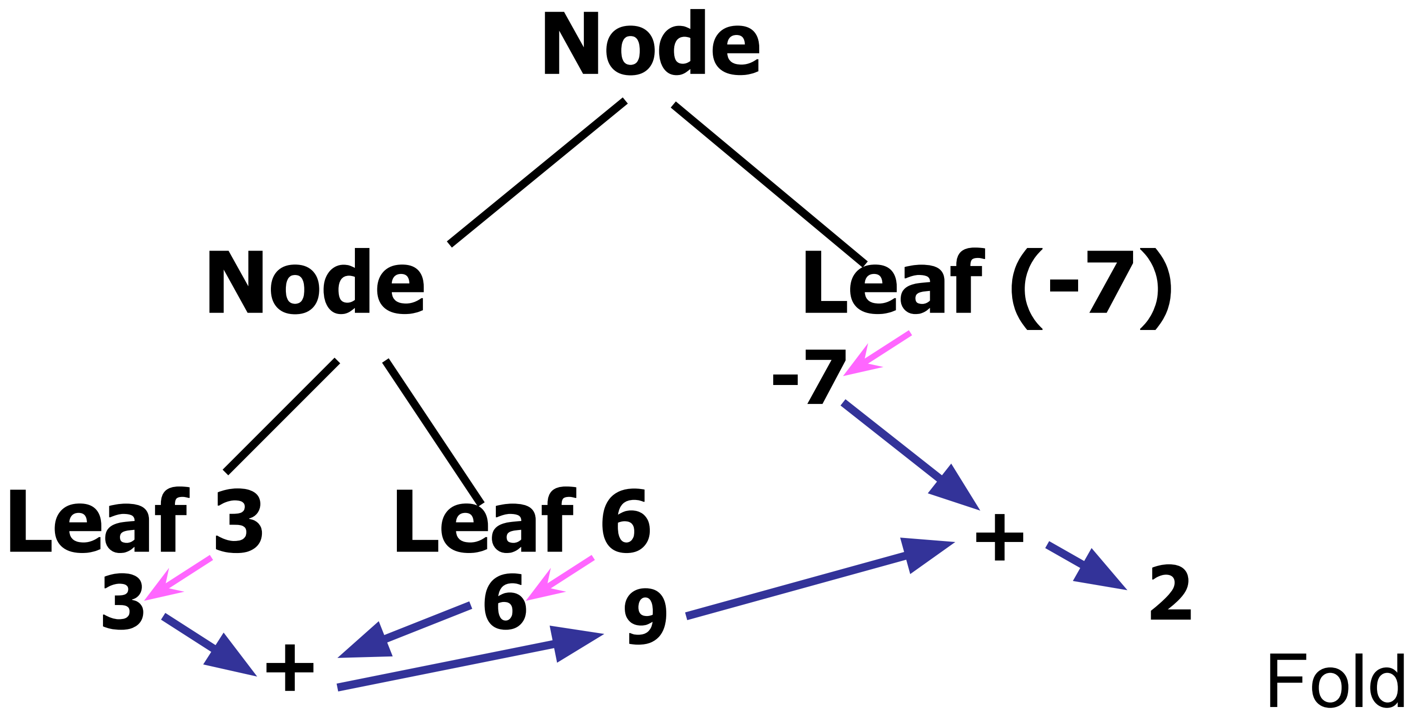
Folding over Recursive Types

```
# let tree_sum = ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum my_tree;;
- : int = 2
```



Folding over Recursive Types

```
# let tree_sum = ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum my_tree;;
- : int = 2
```



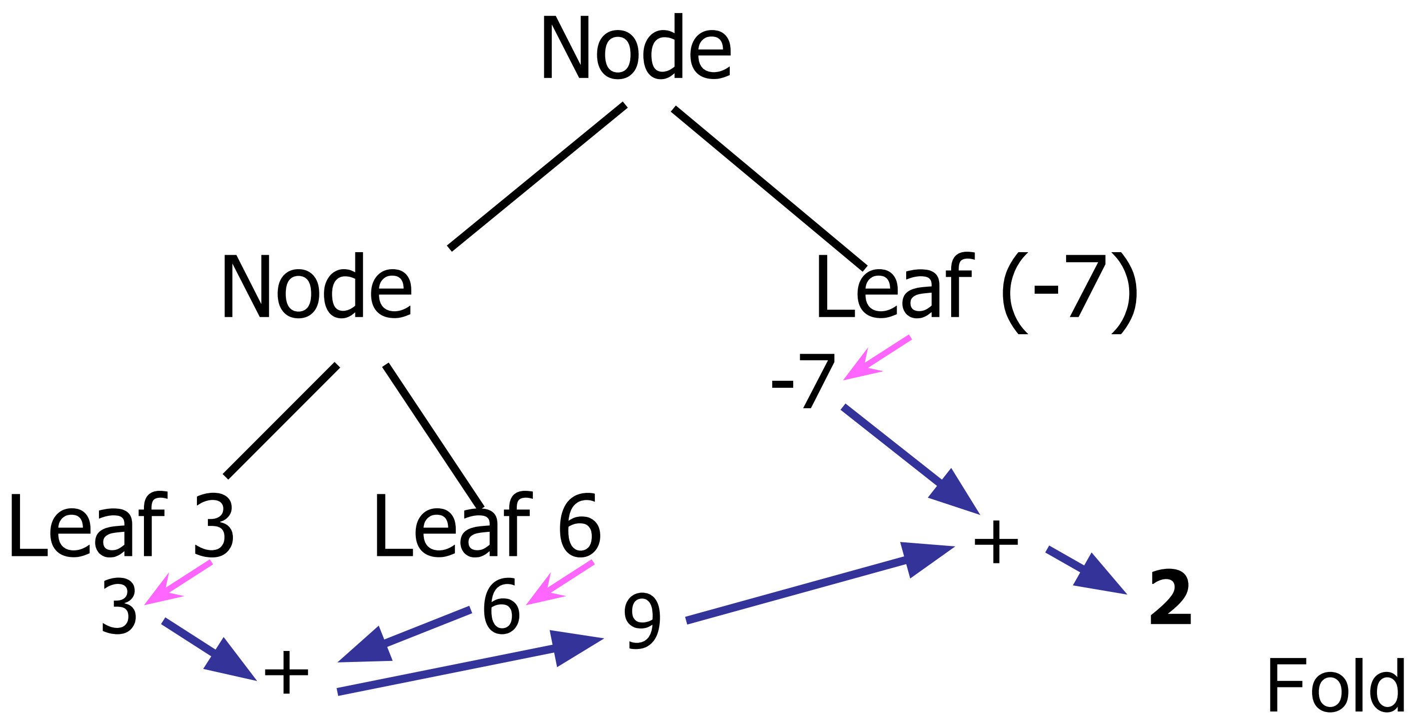
Folding over Recursive Types

let tree_sum = ibtreeFoldRight (fun x -> x) (+);;

val tree_sum : int_Bin_Tree -> int = <fun>

tree_sum my_tree;;

- : int = **2**



Questions so far?

Aside: Folding over ASTs

Aside: Folding over ASTs

Extra credit (EC2) will be about this—will post soon!

Mutually Recursive Datatypes

Mutually Recursive Types

```
type 'a tree =
  TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList =
  Last of 'a tree | More of ('a tree * 'a treeList)
```

Mutually Recursive Types

```
type 'a tree =
  TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList =
  Last of 'a tree | More of ('a tree * 'a treeList)
```

Mutually Recursive Types

```
type 'a tree =
  TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList =
  Last of 'a tree | More of ('a tree * 'a treeList)
```

Mutually Recursive Types

type **'a tree** =
  TreeLeaf of 'a | **TreeNode of 'a treeList**
and 'a treeList =
  Last of 'a tree | More of ('a tree * 'a treeList)

Mutually Recursive Types

type **'a tree** =
  TreeLeaf of 'a | TreeNode of **'a treeList**
and **'a treeList** =
  Last of 'a tree | More of ('a tree * 'a treeList)

Mutually Recursive Types

```
type 'a tree =
  TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList =
  Last of 'a tree | More of ('a tree * 'a treeList)
```

Mutually Recursive Types

type **'a tree** =
  TreeLeaf of 'a | TreeNode of **'a treeList**

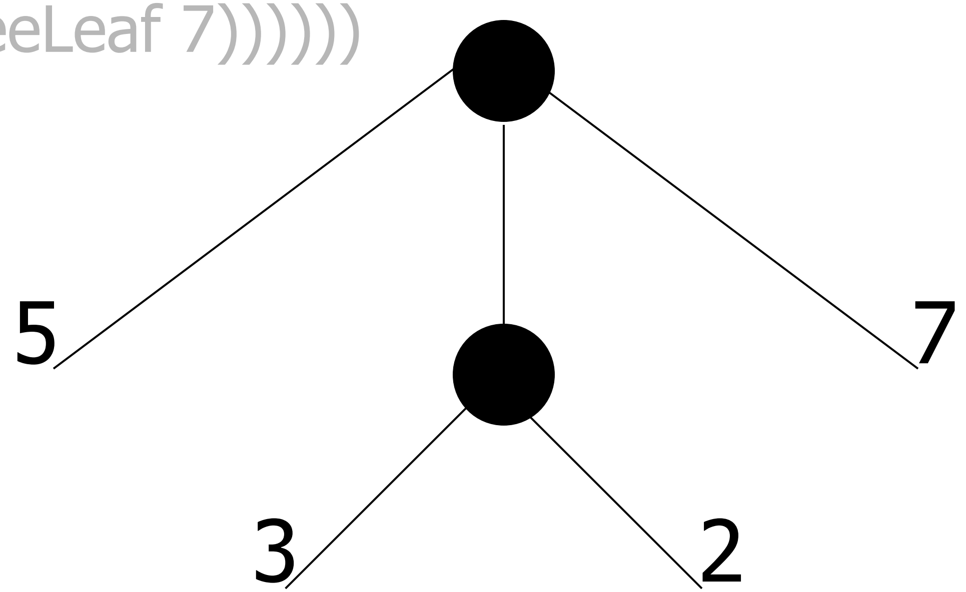
and 'a treeList =
  Last of 'a tree | **More of ('a tree * 'a treeList)**

Mutually Recursive Types

```
type 'a tree =
  TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList =
  Last of 'a tree | More of ('a tree * 'a treeList)
```

Mutually Recursive Types - Values

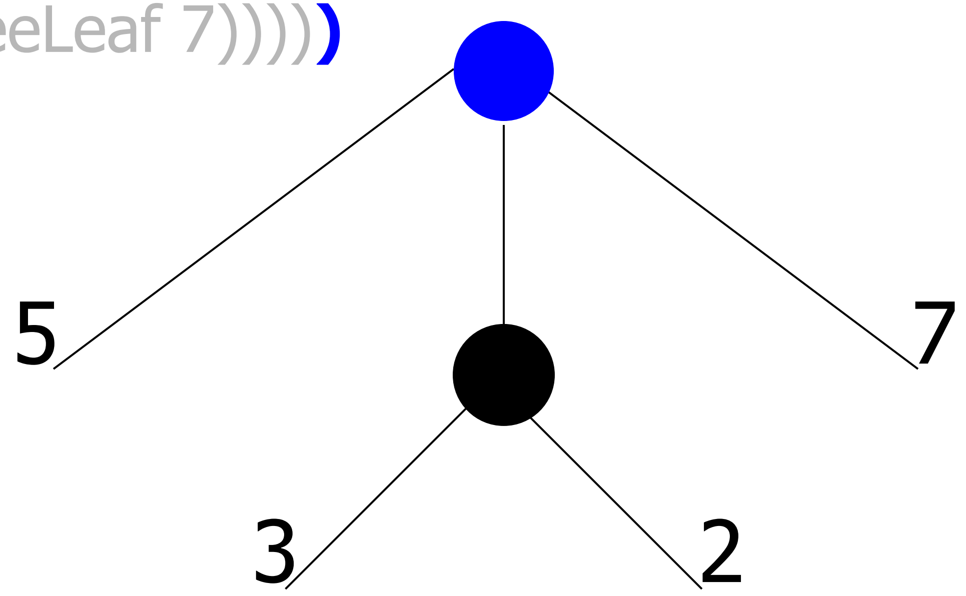
TreeNode
  (More (TreeLeaf 5,
    (More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
      Last (TreeLeaf 7))))))



Mutually Recursive Types - Values

TreeNode

(More (TreeLeaf 5,
  (More
    (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
    Last (TreeLeaf 7)))))



Mutually Recursive Types - Values

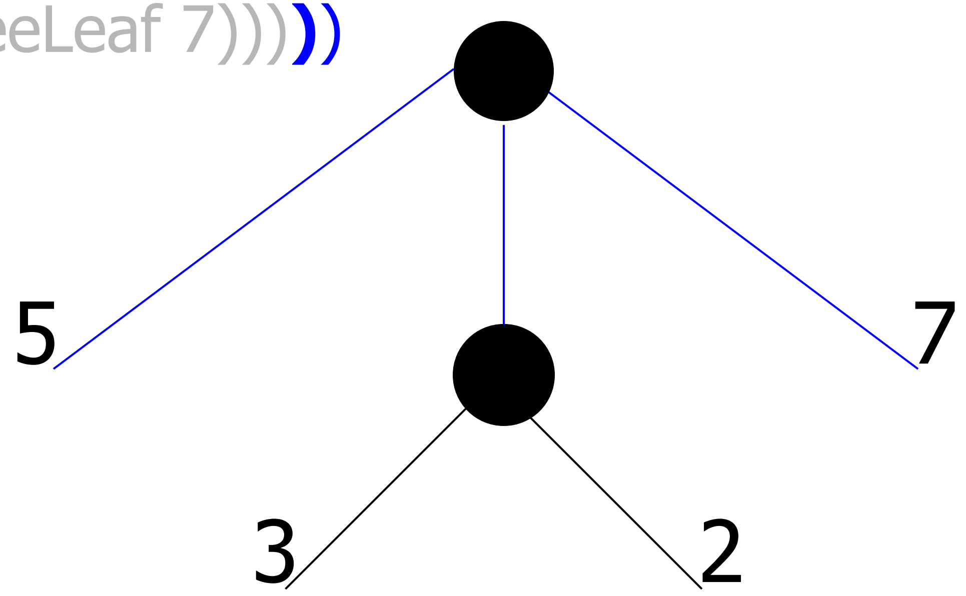
TreeNode

(More (TreeLeaf 5,

(More

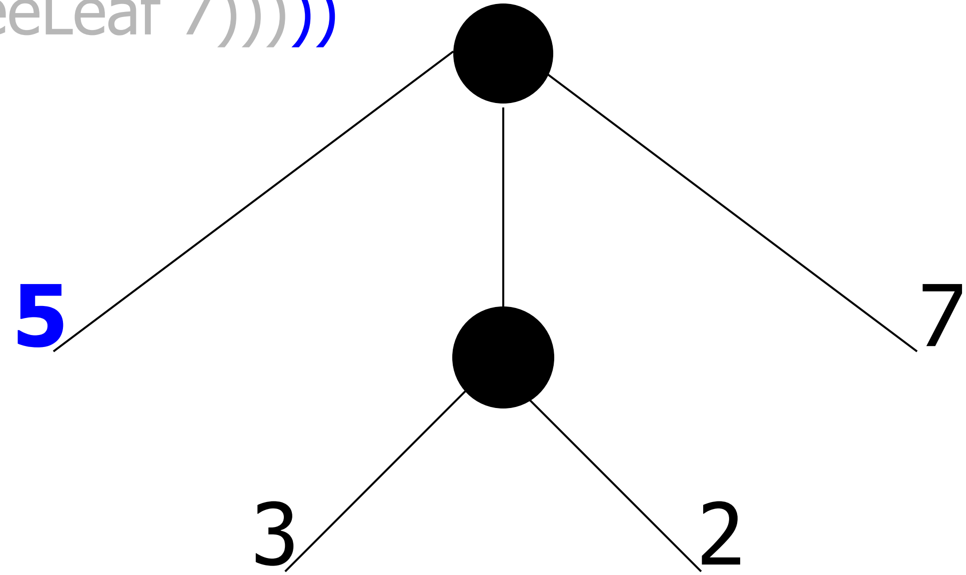
(TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),

Last (TreeLeaf 7))))**))**



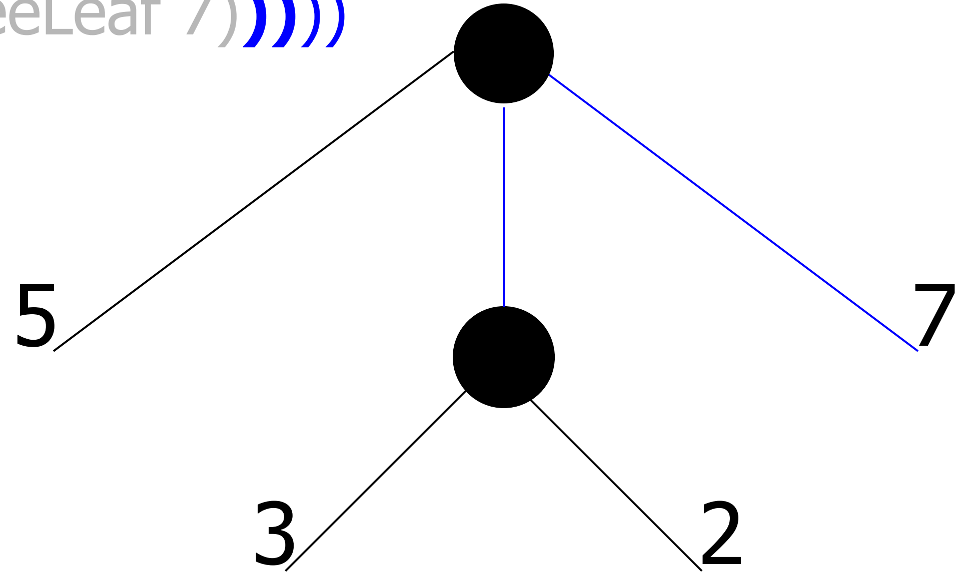
Mutually Recursive Types - Values

TreeNode
 (More (**TreeLeaf 5,**
  (More
   (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
   Last (TreeLeaf 7)))))



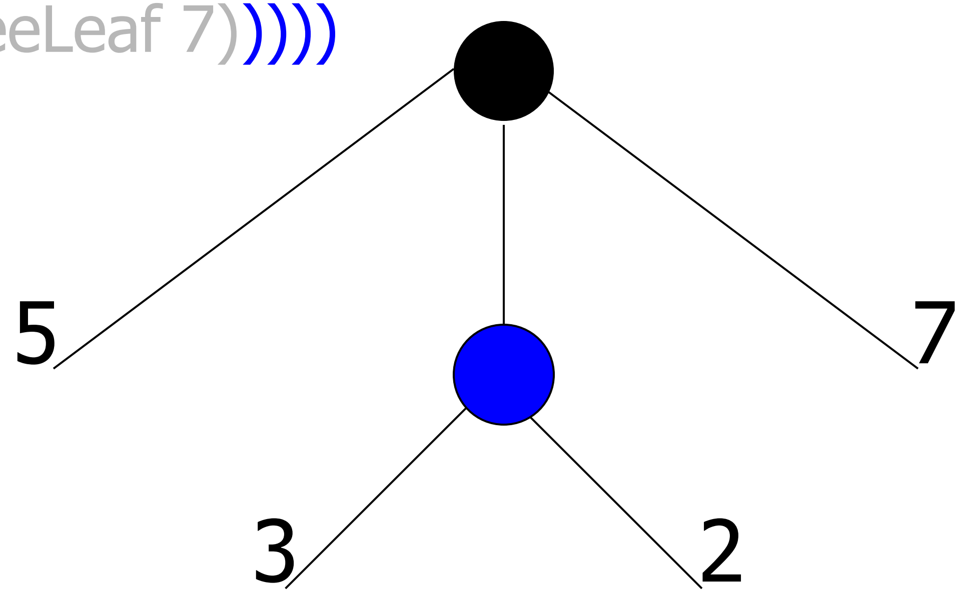
Mutually Recursive Types - Values

TreeNode
  (More (TreeLeaf 5,
    (More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
      Last (TreeLeaf 7)**))))**



Mutually Recursive Types - Values

TreeNode
 (More (TreeLeaf 5,
  (More
   (**TreeNode (**More (TreeLeaf 3, Last (TreeLeaf 2))**),**
   Last (TreeLeaf 7)))))



Mutually Recursive Types - Values

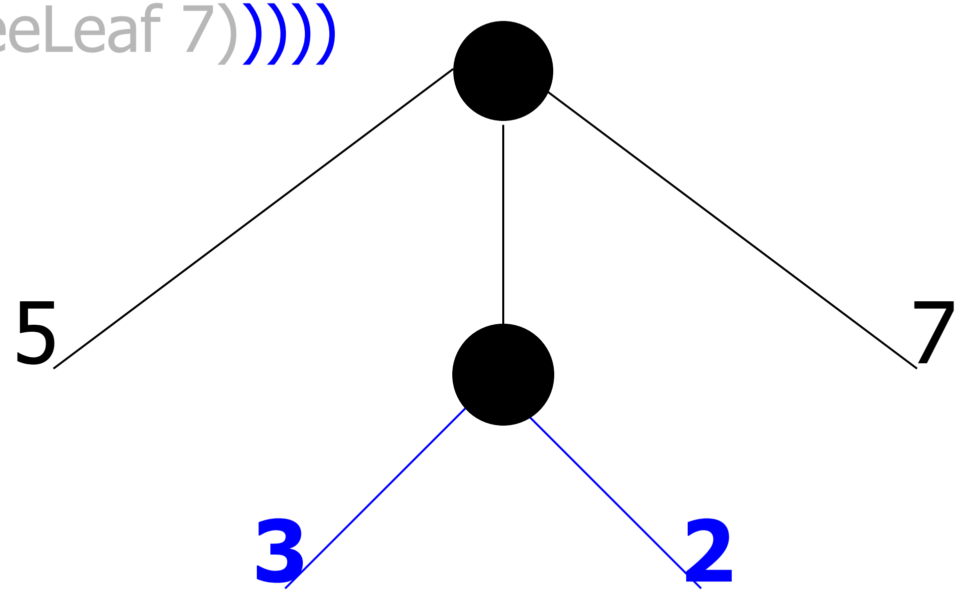
TreeNode

(More (TreeLeaf 5,

(More

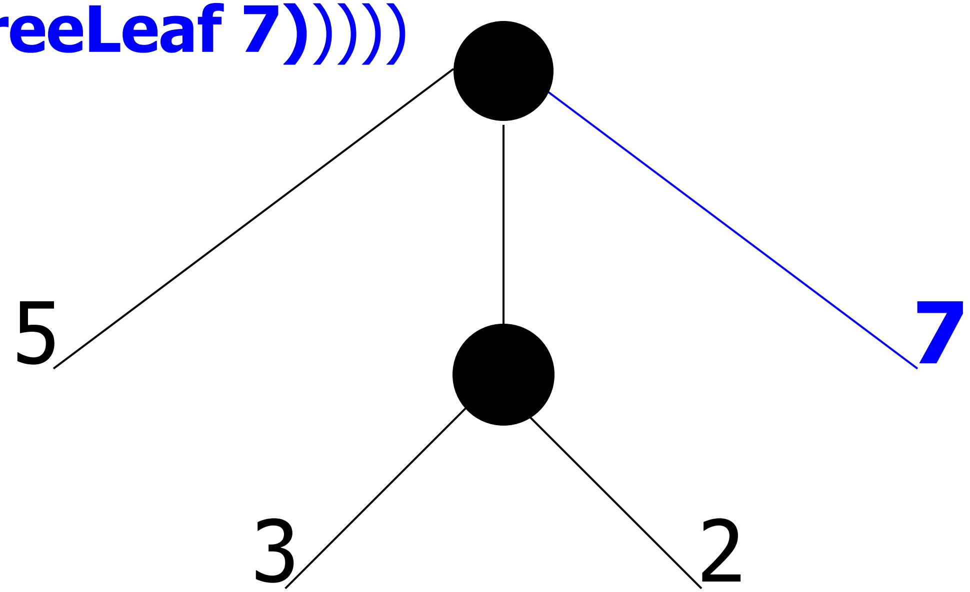
(TreeNode (**More (TreeLeaf 3, Last (TreeLeaf 2))**),

Last (TreeLeaf 7)))))



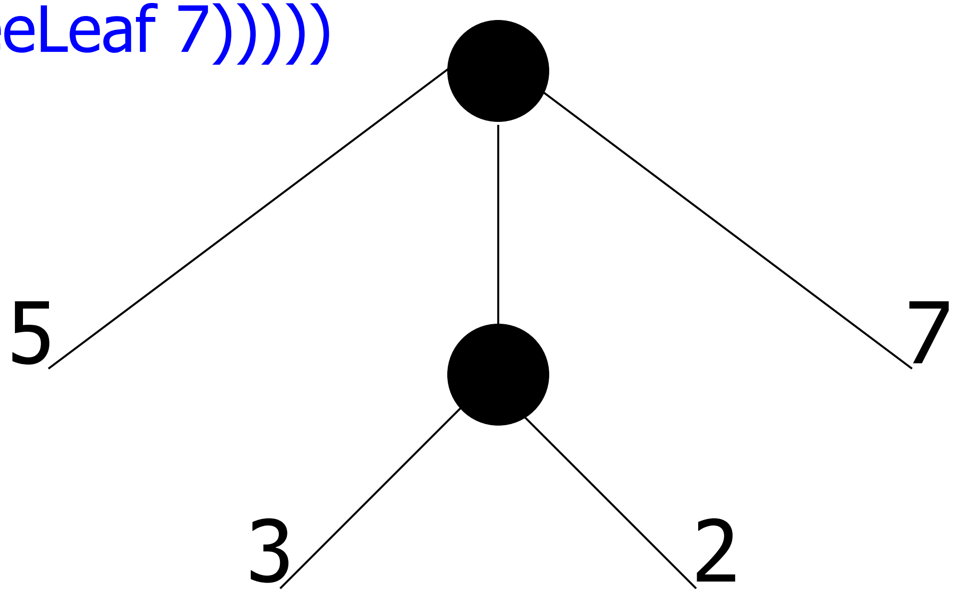
Mutually Recursive Types - Values

TreeNode
  (More (TreeLeaf 5,
    (More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
      Last (TreeLeaf 7)))))

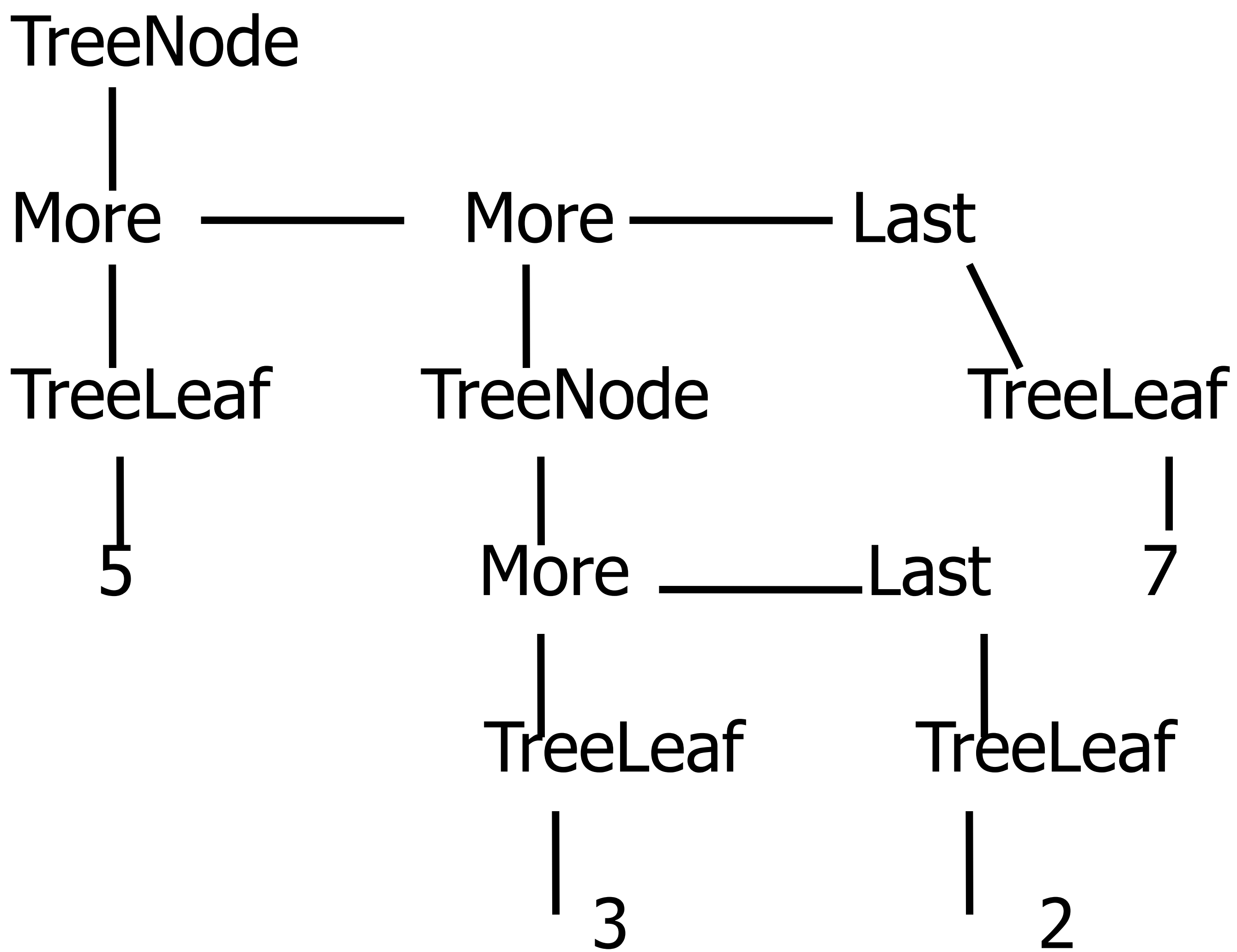


Mutually Recursive Types - Values

TreeNode
  (More (TreeLeaf 5,
    (More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
      Last (TreeLeaf 7)))))

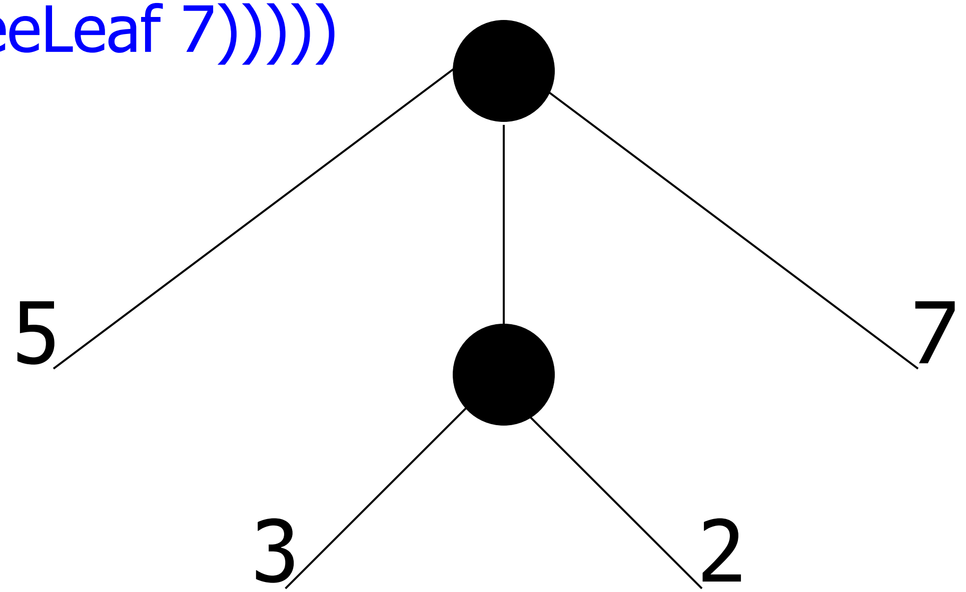


Mutually Recursive Types - Values



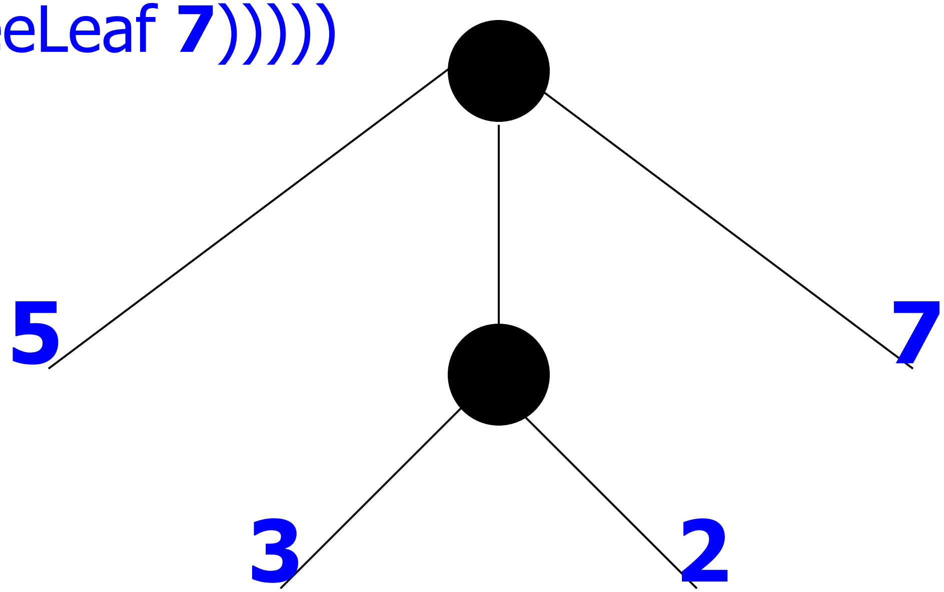
Mutually Recursive **Functions**

TreeNode
  (More (TreeLeaf 5,
    (More
      (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
      Last (TreeLeaf 7)))))



Mutually Recursive **Functions**

TreeNode
  (More (TreeLeaf **5**,
    (More
      (TreeNode (More (TreeLeaf **3**, Last (TreeLeaf **2**))),
      Last (TreeLeaf **7**)))))



Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Functions

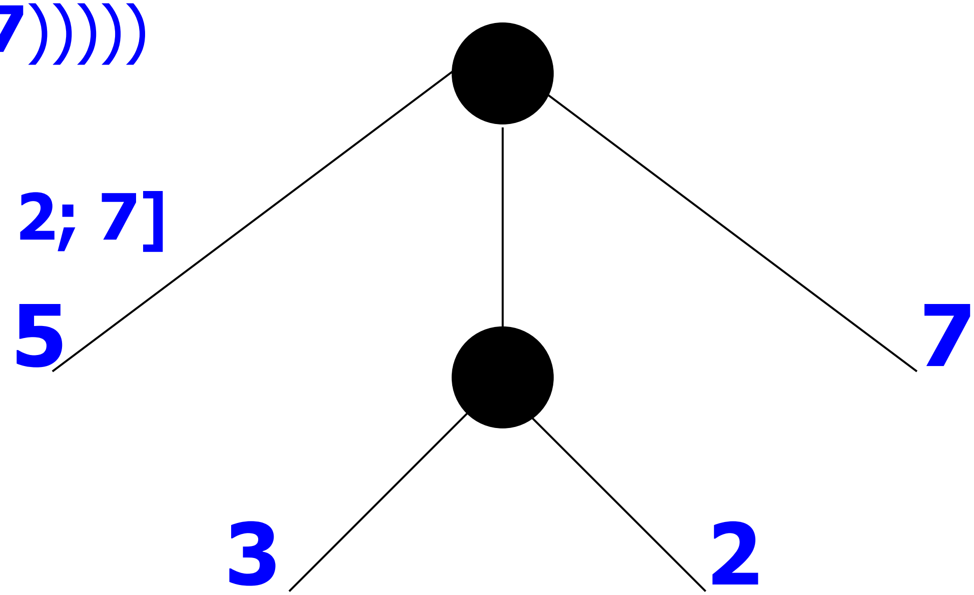
```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive Functions

```
let rec fringe tree =
  match tree with
  | TreeLeaf x -> [x]
  | TreeNode list -> list_fringe list
and list_fringe tree_list =
  match tree_list with
  | Last tree -> fringe tree
  | More (tree, list) ->
      (fringe tree) @ (list_fringe list)
```

Mutually Recursive **Functions**

```
# let tree = TreeNode
   (More (TreeLeaf 5,
     (More
       (TreeNode (More (TreeLeaf 3, Last (TreeLeaf 2))),
       Last (TreeLeaf 7)))))
  in fringe tree;;
 -  : int list = [5; 3; 2; 7]
```



Mutually Recursive Datatypes

Questions so far?

Nested Recursive Datatypes

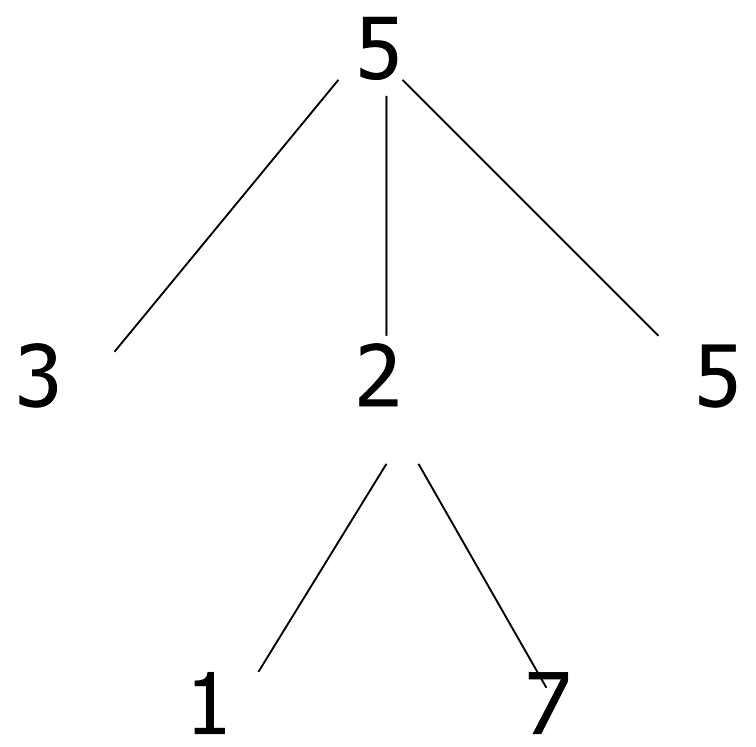
Nested Recursive Types

```
(* Alt. def, allowing empty lists & values anywhere *)
type 'a labeled_tree =
 TreeNode of ('a * 'a labeled_tree list);;
```

Nested Recursive Types - Values

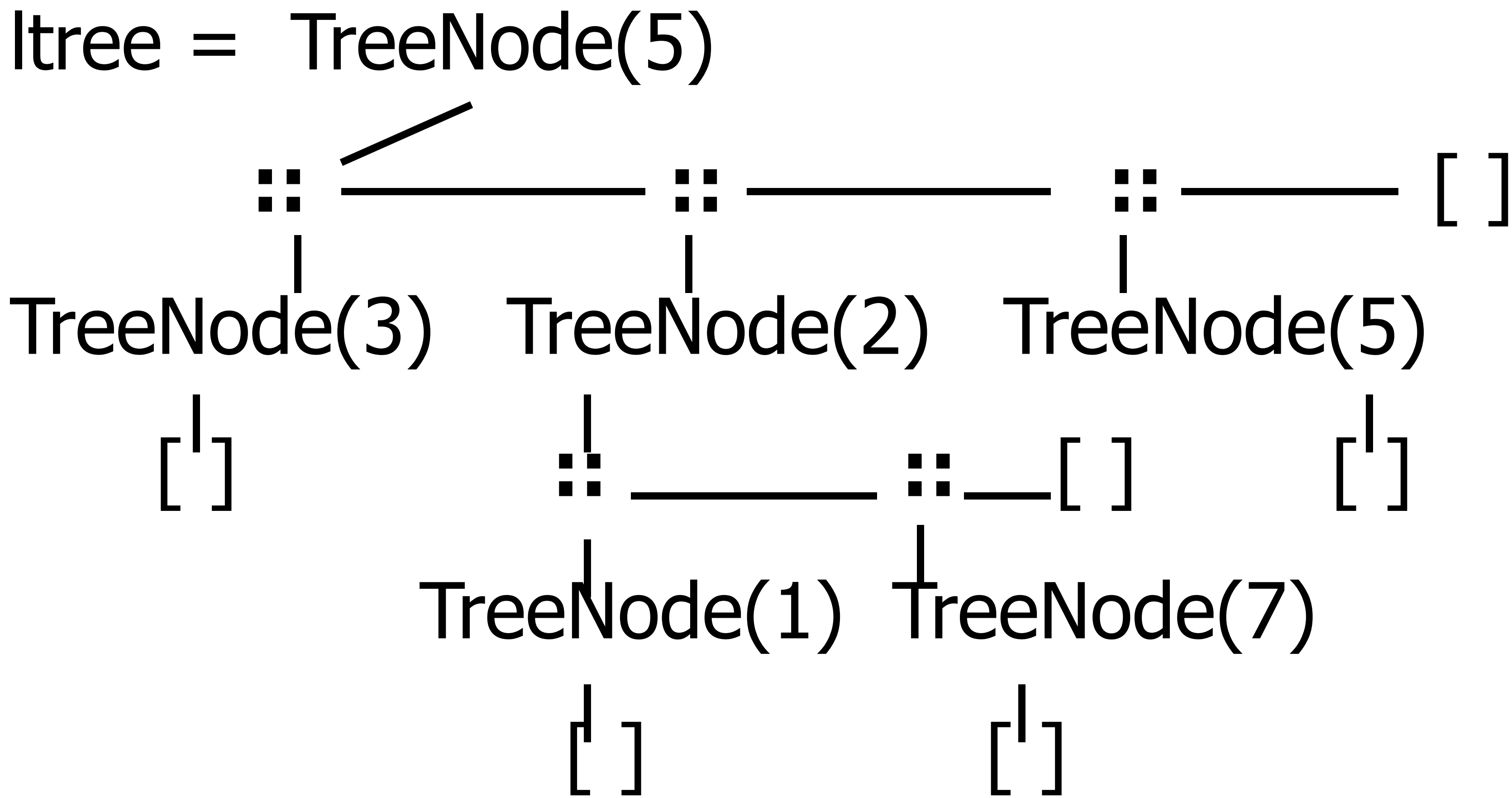
```
(* Alt. def, allowing empty lists & values anywhere *)
type 'a labeled_tree =
 TreeNode of ('a * 'a labeled_tree list);;

TreeNode
  (5,
    [TreeNode (3, []);
     TreeNode
       (2, [TreeNode (1, []); TreeNode (7, [])]);
     TreeNode (5, [])])
```

Nested Recursive Types - Values




```
TreeNode
  (5,
    [TreeNode (3, []);
     TreeNode
       (2, [TreeNode (1, []); TreeNode (7, [])]);
     TreeNode (5, [])])
```

Nested Recursive Types - Values



Mutually Recursive Functions

```
let rec flatten_tree labtree =
  match labtree with
  | TreeNode (x, ts) -> x :: flatten_tree_list ts
and flatten_tree_list ts =
  match ts with
  | [] -> []
  | labtree :: labtrees ->
    flatten_tree labtree @ flatten_tree_list labtrees
```

Mutually Recursive Functions

```
let rec flatten_tree labtree =
  match labtree with
  | TreeNode (x, ts) -> x :: flatten_tree_list ts
and flatten_tree_list ts =
  match ts with
  | [] -> []
  | labtree :: labtrees ->
    flatten_tree labtree @ flatten_tree_list labtrees
```

Nested recursive types lead to **mutually** recursive functions!

Questions?

Takeaways

- We saw **three kinds of datatypes:**
  - **recursive**
  - **mutually recursive**
  - **nested recursive**
- All useful for representing **language syntax**
- **Functions** over these datatypes can **reason** about program syntax, **interpret** programs (implicitly defini a **semantics**), **transform** programs, etc.
- **Recursive** types -> **recursive** functions
- **Mutually** recursive types -> **mutual** recursion
- **Nested** recursive types -> **mutual** recursion, too

Next Class

- **Will grade EC1 soon!**
- **Will post EC2 soon!**
- **MP4** will be due next Tuesday
- **WA4** will be due next Thursday
- All deadlines can be found on **course website**
- Use **office hours** and **class forums** for help