Programming Languages and Compilers (CS 421)

William Mansky

http://courses.engr.illinois.edu/cs421/su2013/

Based in part on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, Elsa Gunter, and Dennis Griffith

Variants – Algebraic Data Types

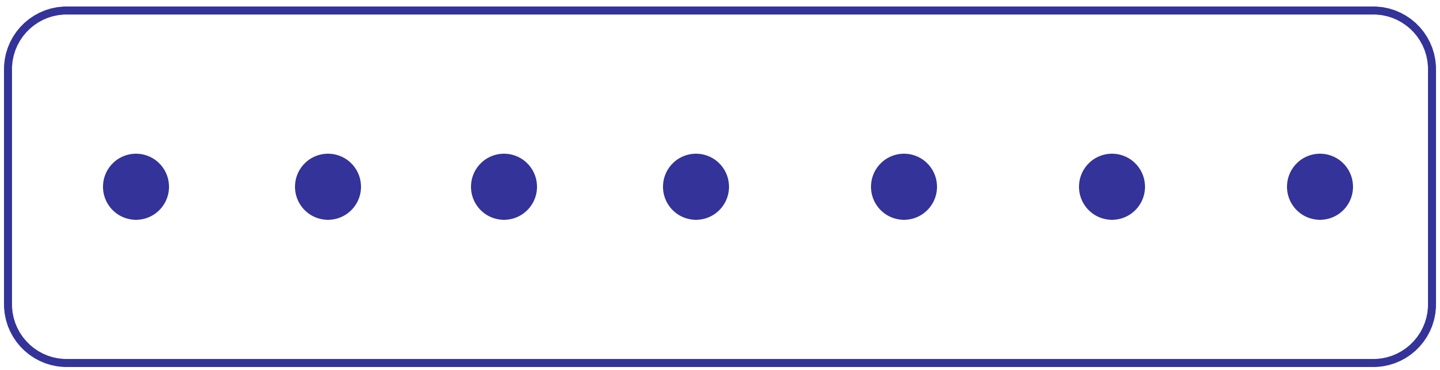
- Core of user-defined types in Ocaml
- Support enumerations, disjoint unions, recursive types
- Write functions with pattern matching, recursion
- Already seen one example: lists

Variants - Syntax (slightly simplified)

- type *name* = C_1 [of ty_1] | . . . | C_n [of ty_n]
- Introduce a type called *name*
- C_1 : ty_1 -> *name*
- C_i is called a *constructor*; if the optional type argument is omitted, it is called a *constant*
- Constructors are the basis of almost all pattern matching

Enumeration Types as Variants

An enumeration type is a collection of distinct values



They are ordered by their declaration order

Enumeration Types as Variants

```
# type weekday = Monday | Tuesday | Wednesday
    | Thursday | Friday | Saturday | Sunday;;
type weekday =
    Monday
  | Tuesday
  | Wednesday
  | Thursday
  | Friday
  | Saturday
  | Sunday
```

Functions over Enumerations

```
# let day_after day = match day with
    Monday -> Tuesday
  | Tuesday -> Wednesday
  | Wednesday -> Thursday
  | Thursday -> Friday
  | Friday -> Saturday
  | Saturday -> Sunday
  | Sunday -> Monday;;
 val day_after : weekday -> weekday = <fun>
```

Functions over Enumerations

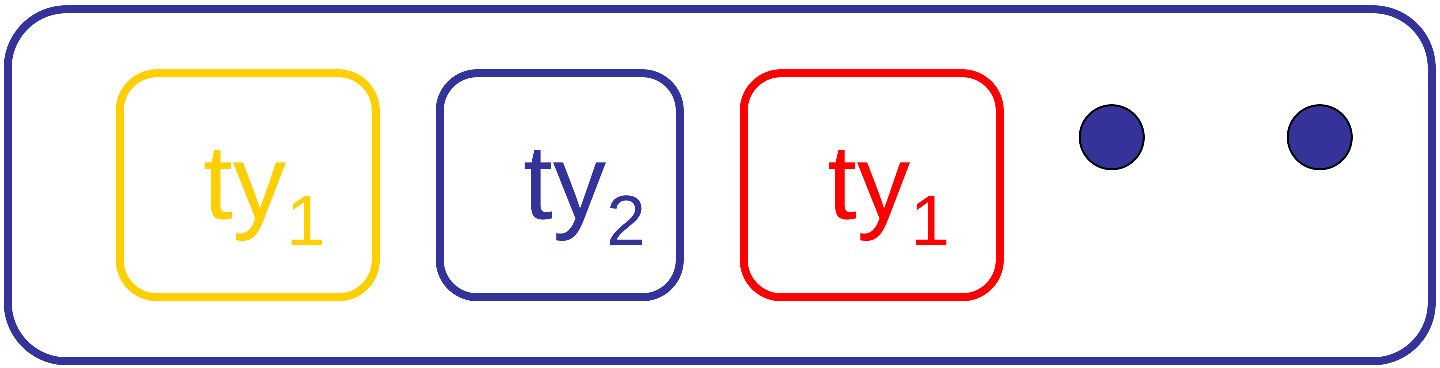
```
# let rec days_later n day =
    match n with 0 -> day
    | _ -> if n > 0
          then day_after (days_later (n - 1) day)
          else days_later (n + 7) day;;
val days_later : int -> weekday -> weekday
  = <fun>
```

Functions over Enumerations

```
# days_later 2 Tuesday;;
- : weekday = Thursday
# days_later (-1) Wednesday;;
- : weekday = Tuesday
# days_later (-4) Monday;;
- : weekday = Thursday
```

Disjoint Union Types

- Disjoint union of types, with some possibly occurring more than once



- We can also add in some new singleton elements

Disjoint Union Types

```
# type id = DriversLicense of int
  | SocialSecurity of int | Name of string;;
# let x = DriversLicense 123;;
val x : id = DriversLicense 123
# let check_id id = match id with
      DriversLicense num ->
       not (List.mem num [13570; 99999])
    | SocialSecurity num -> num < 900000000
    | Name str -> not (str = "John Doe");;
 val check_id : id -> bool = <fun>
```

Polymorphism in Variants

- The type 'a option is gives us something to represent non-existence or failure

```
# type 'a option = Some of 'a | None;;
type 'a option = Some of 'a | None
```

- Used to encode partial functions
- Often can replace the raising of an exception

Functions over option

```
# let rec first p list =
    match list with [ ] -> None
    | (x::xs) -> if p x then Some x else first p xs;;
val first : ('a -> bool) -> 'a list -> 'a option = <fun>
# first (fun x -> x > 3) [1;3;4;2;5];;
- : int option = Some 4
# first (fun x -> x > 5) [1;3;4;2;5];;
- : int option = None
```

Mapping over Variants

```
# let optionMap f opt =
    match opt with None -> None
    | Some x -> Some (f x);;
val optionMap : ('a -> 'b) -> 'a option -> 'b
  option = <fun>
# optionMap
  (fun x -> x - 2)
  (first (fun x -> x > 3) [1;3;4;2;5]);;
- : int option = Some 2
```

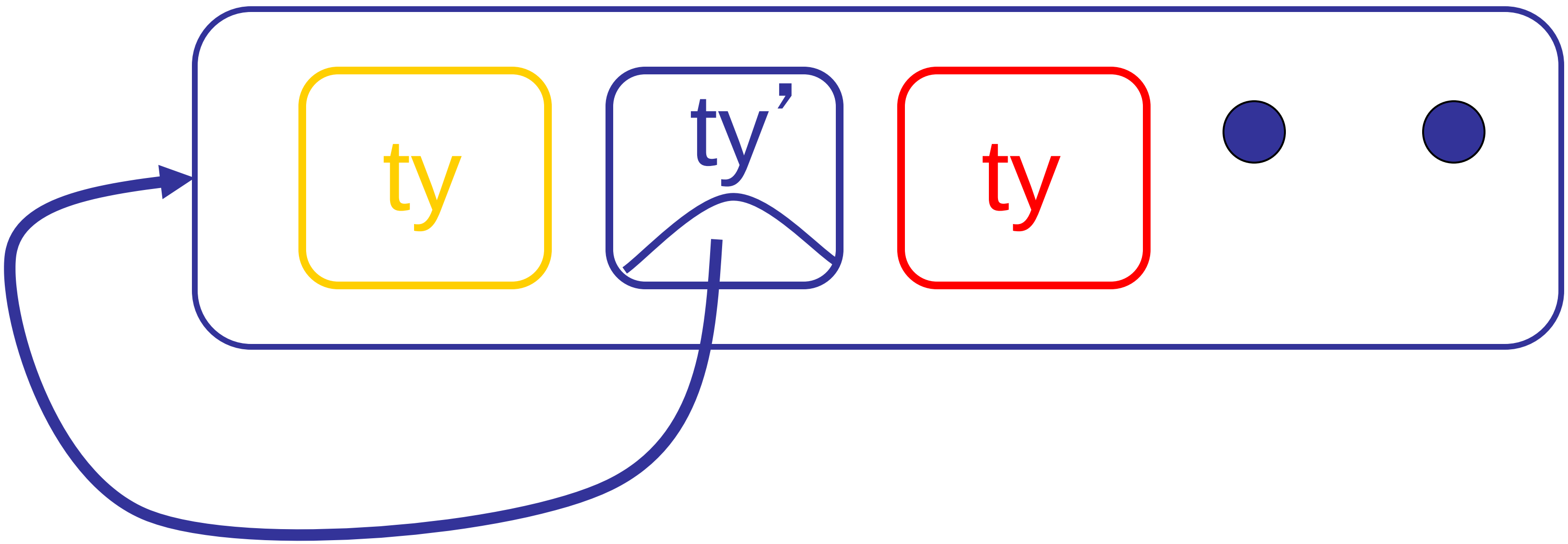
Folding over Variants

```
# let optionFold someFun noneVal opt =
    match opt with None -> noneVal
    | Some x -> someFun x;;
val optionFold : ('a -> 'b) -> 'b -> 'a option ->
  'b = <fun>

# let optionMap f opt =
    optionFold (fun x -> Some (f x)) None opt;;
val optionMap : ('a -> 'b) -> 'a option -> 'b
  option = <fun>
```

Recursive Types

- The type being defined may be a component of itself



Recursive Type Example 1: Lists

- type ‘a mylist = Nil | Cons of (‘a * ‘a mylist)
- Real lists use nicer syntax, but have the same behavior

Recursive Type Example 2: Trees

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

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

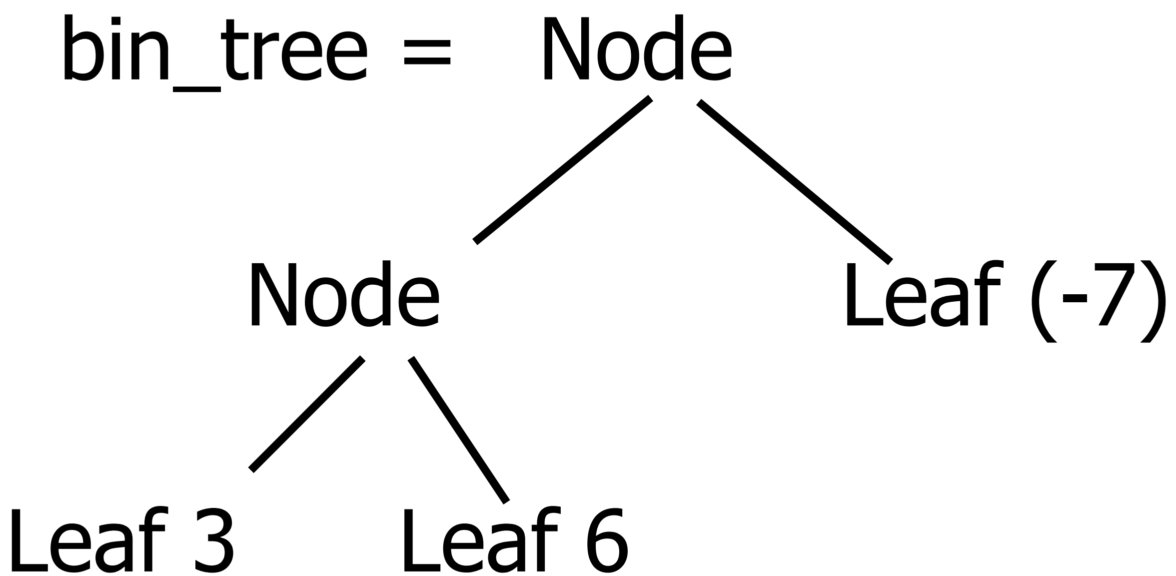
Recursive Data Type Values

```
# let bin_tree =
 Node (Node (Leaf 3, Leaf 6),Leaf (-7));;

val bin_tree : int_Bin_Tree = Node (Node
  (Leaf 3, Leaf 6), Leaf (-7))
```

Recursive Data Type Values

let bin_tree =
 Node (Node (Leaf 3, Leaf 6),Leaf (-7));;



Recursive Functions

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

Mapping over Recursive Types

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

Mapping over Recursive Types

```
# ibtreeMap ((+) 2) bin_tree;;

- : int_Bin_Tree = Node (Node (Leaf 5, Leaf
  8), Leaf (-5))
```

Folding over Recursive Types

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

Folding over Recursive Types

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

General Folding

- Replace constructors with functions that take recursively computed values
- Gives a bottom up traversal like fold_right
- Extra work to do top down (fold_left)

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);;
```

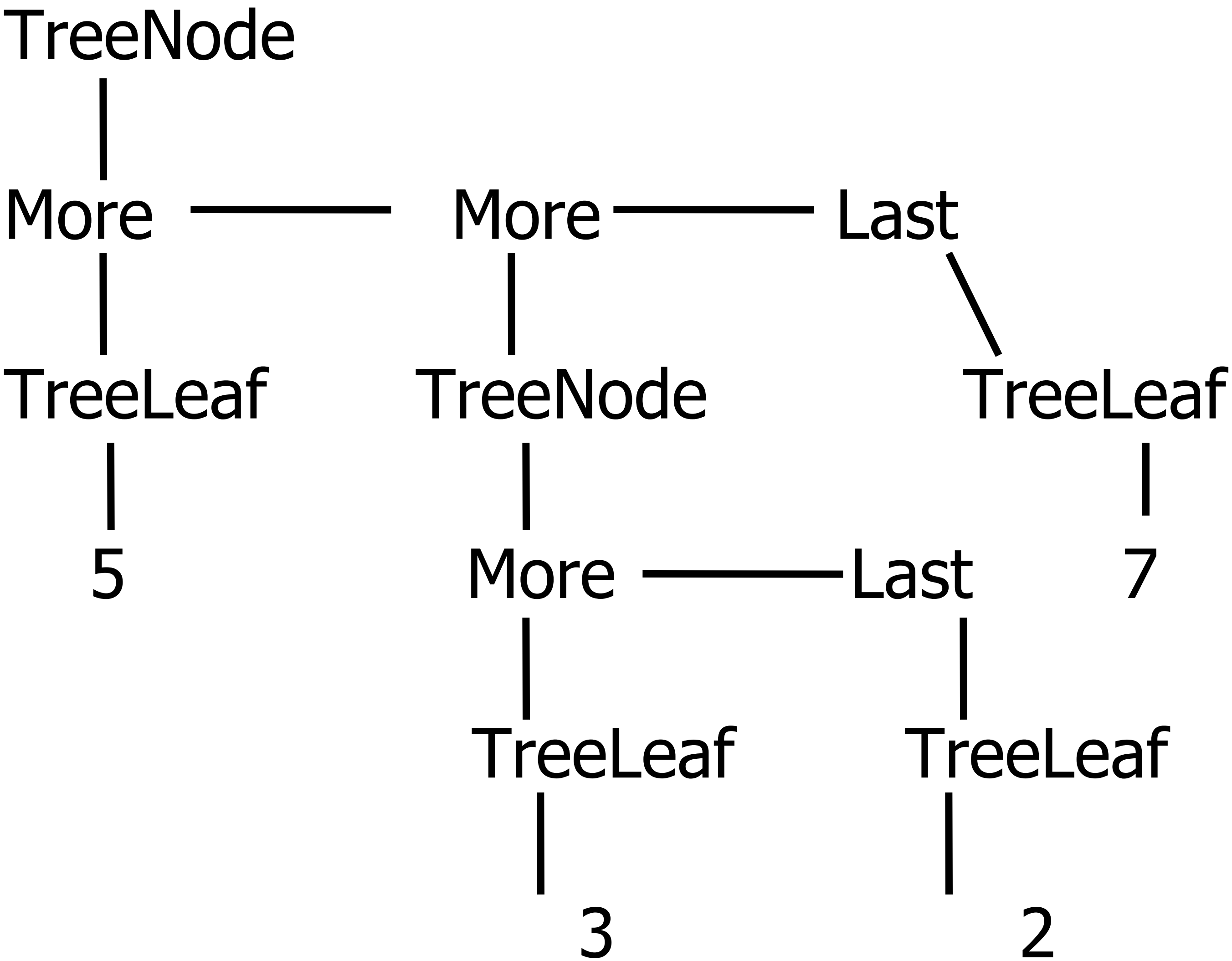
type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList

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

Mutually Recursive Types - Values

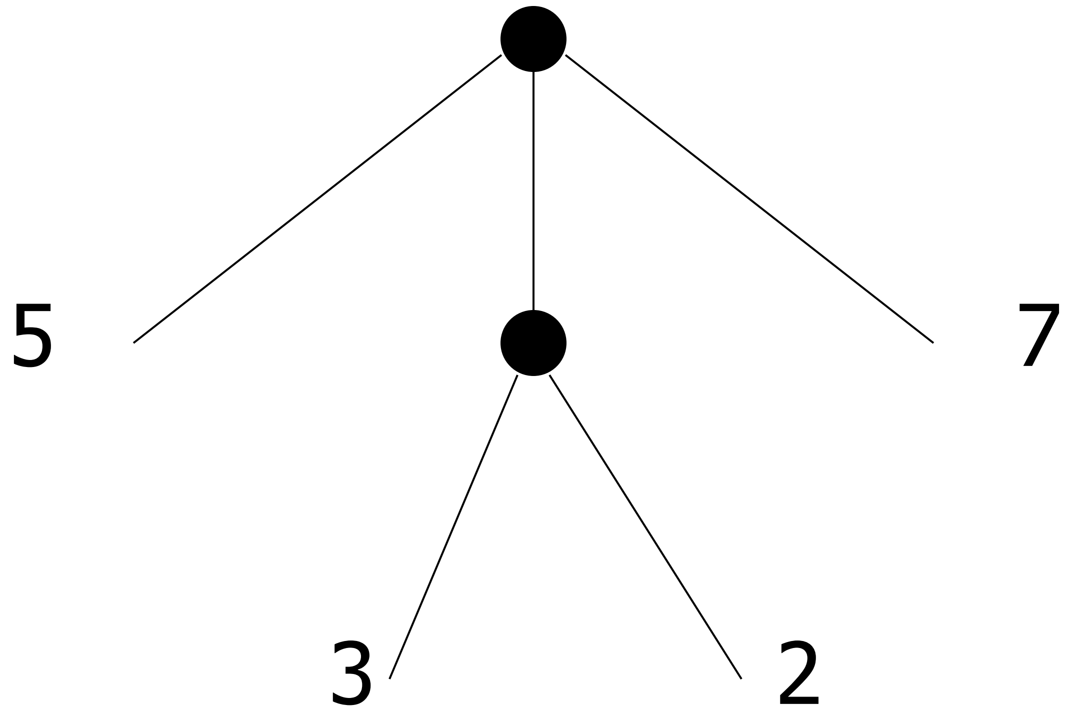
```
# let tree =
  TreeNode
   (More (TreeLeaf 5,
         (More (TreeNode
               (More (TreeLeaf 3,
                     Last (TreeLeaf 2))),
               Last (TreeLeaf 7)))));;
```

Mutually Recursive Types - Values



Mutually Recursive Types - Values

A more conventional picture



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);;

val fringe : 'a tree -> 'a list = <fun>
val list_fringe : 'a treeList -> 'a list = <fun>
```

Mutually Recursive Functions

fringe tree;;

- : int list = [5; 3; 2; 7]

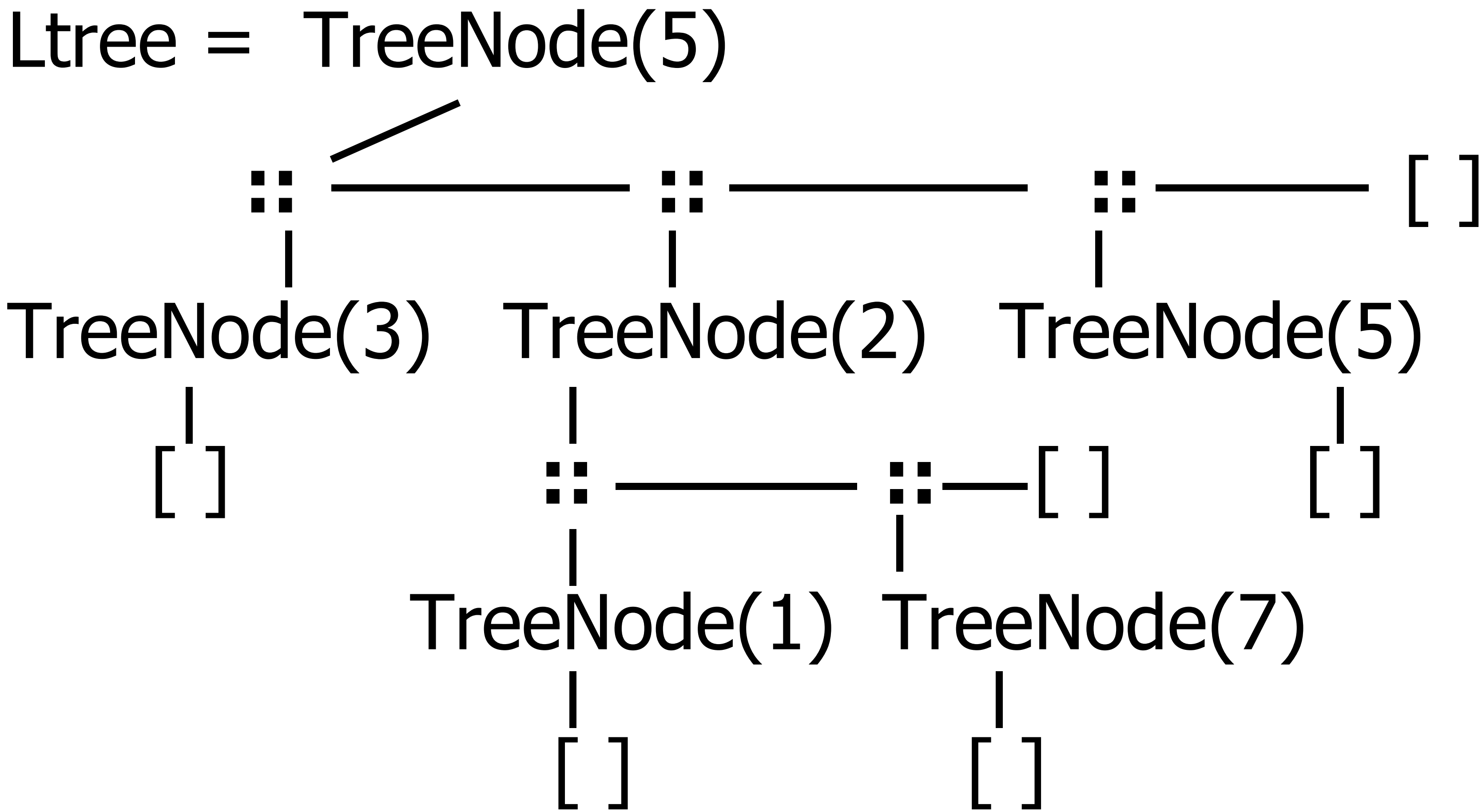
Nested Recursive Types

```
# type 'a labeled_tree =
 TreeNode of ('a * 'a labeled_tree
  list);;
type 'a labeled_tree = TreeNode of ('a
  * 'a labeled_tree list)
```

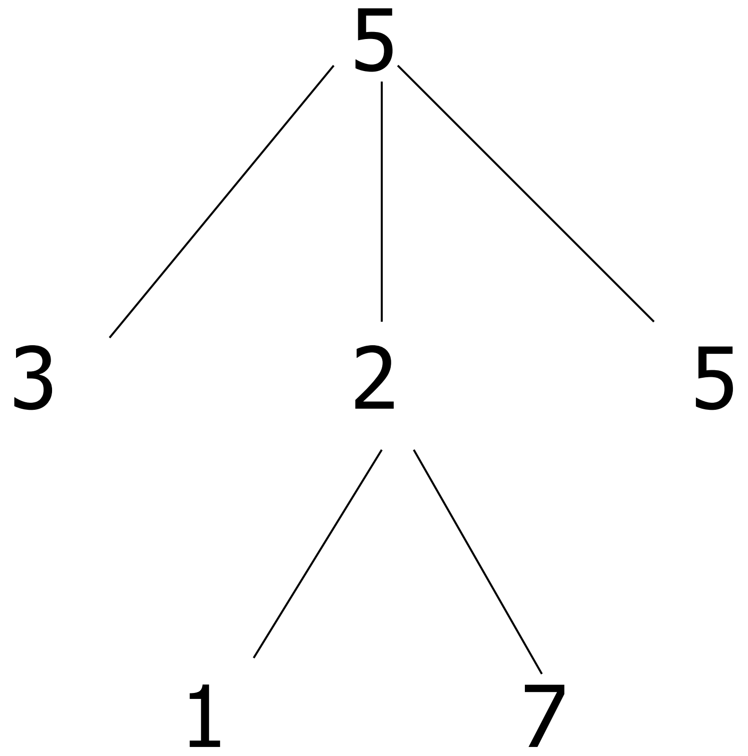
Nested Recursive Type Values

```
# let ltree =
  TreeNode (5,
    [TreeNode (3, []);
     TreeNode (2, [TreeNode (1, []);
                   TreeNode (7, [])]);
     TreeNode (5, [])]);;
```

Nested Recursive Type Values



Nested Recursive Type Values



Mutually Recursive Functions

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

Mutually Recursive Functions

val flatten_tree : 'a labeled_tree -> 'a list =
<fun>

val flatten_tree_list : 'a labeled_tree list -> 'a list = <fun>

flatten_tree ltree;;

- : int list = [5; 3; 2; 1; 7; 5]

- Nested recursive types lead to mutually recursive functions

Mutually Recursive Functions

\# flatten_tree (TreeNode (5, [TreeNode (1, []); TreeNode (2, [])]));;

5 :: flatten_tree_list [TreeNode (1, []); TreeNode (2, [])]

5 :: (flatten_tree (TreeNode (1, []))) @ (flatten_tree_list ([TreeNode (2, [])]))

[5; 1] @ (flatten_tree_list ([TreeNode (2, [])]))

[5; 1] @ flatten_tree (TreeNode (2, [])) @ []

[5; 1] @ [2] @ [] = [5; 1; 2]

Infinite Recursive Values

```
# let rec ones = 1::ones;;
val ones : int list =
  [1; 1; 1; 1; ...]

# match ones with x::xs -> x;;
Warning: ...
- : int = 1
```

Infinite Recursive Values

```
# let rec ones = 1::ones;;
val ones : int list =
  [1; 1; 1; 1; ...]

# let other_ones = match ones with x::xs ->
   xs;;
Warning: ...
- : int list = [1; 1; 1; 1; ...]

# other_ones = ones;;
(* runs forever – don’t do this! *)
```

Infinite Recursive Values

```
# let rec lab_tree = TreeNode(2, tree_list)
   and tree_list = [lab_tree; lab_tree];;
val lab_tree : int labeled_tree =
  TreeNode (2, [TreeNode(...); TreeNode(...)])
val tree_list : int labeled_tree list =
  [TreeNode (2, [TreeNode(...);
   TreeNode(...)]);
   TreeNode (2, [TreeNode(...);
   TreeNode(...)])]
```

Infinite Recursive Values

```
# match lab_tree
  with TreeNode (x, _) -> x;;
- : int = 2
```

Records

- Records serve the same programming purpose as tuples
- Provide better documentation, more readable code
- Allow components to be accessed by label instead of position
  - Labels (aka *field names* must be unique)
  - Fields accessed by suffix dot notation

Record Types

- Record types must be declared before they can be used in OCaml

type person = {name : string; ss : (int * int * int); age : int};;

type person = { name : string; ss : int * int * int; age : int; }

- person is the type being introduced
- name, ss and age are the labels, or fields

Record Values

- Records built with labels; order does not matter

```
# let teacher = {name = "Elsa L. Gunter";
  age = 102; ss = (119,73,6244)};;
val teacher : person =
 {name = "Elsa L. Gunter"; ss = (119, 73,
  6244); age = 102}
```

Record Values

```
# let student = {ss=(325,40,1276);
  name="Joseph Martins"; age=22};;
val student : person =
  {name = "Joseph Martins"; ss = (325, 40,
  1276); age = 22}
# student = teacher;;
- : bool = false
```

Record Pattern Matching

```
# let {name = elsa; age = age; ss =
  (_,_,s3)} = teacher;;
val elsa : string = "Elsa L. Gunter"
val age : int = 102
val s3 : int = 6244
```

Record Field Access

```
# let soc_sec = teacher.ss;;
val soc_sec : int * int * int = (119,
  73, 6244)
```

New Records from Old

```
# let birthday person = {person with age =
  person.age + 1};;
val birthday : person -> person = <fun>
# birthday teacher;;
- : person = {name = "Elsa L. Gunter"; ss =
  (119, 73, 6244); age = 103}
```

New Records from Old

```
# let new_id name soc_sec person =
 {person with name = name; ss = soc_sec};;
val new_id : string -> int * int * int -> person
  -> person = <fun>
# new_id "Giuseppe Martin" (523,04,6712)
  student;;
- : person = {name = "Giuseppe Martin"; ss =
  (523, 4, 6712); age = 22}
```