Programming Languages and Compilers (CS 421)

#9 and #10: Algebraic datatypes; disjoint union types, product types, recursive datatypes

Madhusudan Parthasarathy

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

Based on slides by Elsa Gunter, which in turn is partly based on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Midterm

- Midterm from Oct 2 – Oct 4
  - CBTF
  - Topics: All topics covered till Thu Sep 20, which includes writing functions in CPS form
  - Mostly all that you have done (WAs, MPs, MLs), but will include extra questions
  - More details on Piazza soon, including practice exam.

Midterm

Studying for this exam

- Understand the lecture slides and discussions thoroughly.
- Revisit the MPs, MLs and WAs and make sure you understand the solutions thoroughly. Repeat any you are not comfortable with.
- Take the pdf sample exam as a thorough overview for the actual exam.
- Take the PrairieLearn Midterm1 Practice to be familiar with the precise nature of the questions and to see where you may have trouble taking the test in a timely enough manner.

Midterm

Syllabus: First 8 lectures (till Sep 20);

all videos are online at echo360.org; slides are up to date

Basic OCaml

- Know the basic constructs (e.g., match, fun, let, let rec) like the back of your hand.
- Be able to determine the type of OCaml expressions
- Be able to evaluate OCaml expressions, both intuitively, and and step by step followong the steps discussed in class
- Be able to describe the environment that results from a sequence of declarations
- Be able to describe the closure that is the result of evalutating a function declaration
- Understand what effect sequencing, function application and lambda lifting has on the order of evaluation of expressions

Midterm

Recursion

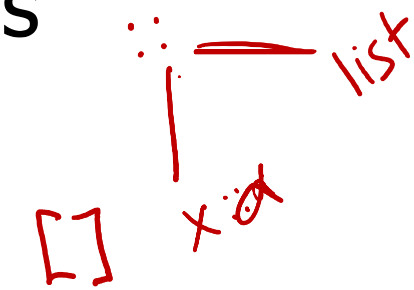
- Be able to write recursive functions, including (but not necessarily limited to) tail-recursive or forward recursive.
- Be able to recognize whether a function is tail-recursive, and when a recursive call is in tail call position

Higher Order Functions (HOFs)

- Be able to write the definitions of the common HOFs.
- Be able to use map and fold to implement other functions, as in ML2.
- Be able to write functions that use other functions as arguments
- Continuations and Continuation Passing Style
- Understand what the basic idea of what a continuation is.
- Be able rewrite an operation / procedure in direct style to take a continuation to which to pass its results, while preserving the order of evaluation.
- Be able to put a complex, possibly recursive procedure into full continutation passing style, while preserving the order of evaluation.

Data type in Ocaml: lists

- Frequently used lists in recursive program
- Matched over two structural cases
  - [ ] - the empty list
  - (x :: xs) a non-empty list

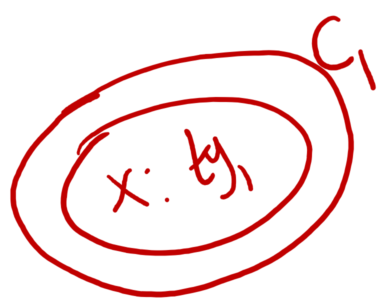


- Covers all possible lists
- type 'a list = [ ] | (::) of 'a * 'a list
  - Not quite legitimate declaration because of special syntax

[5;2;3]

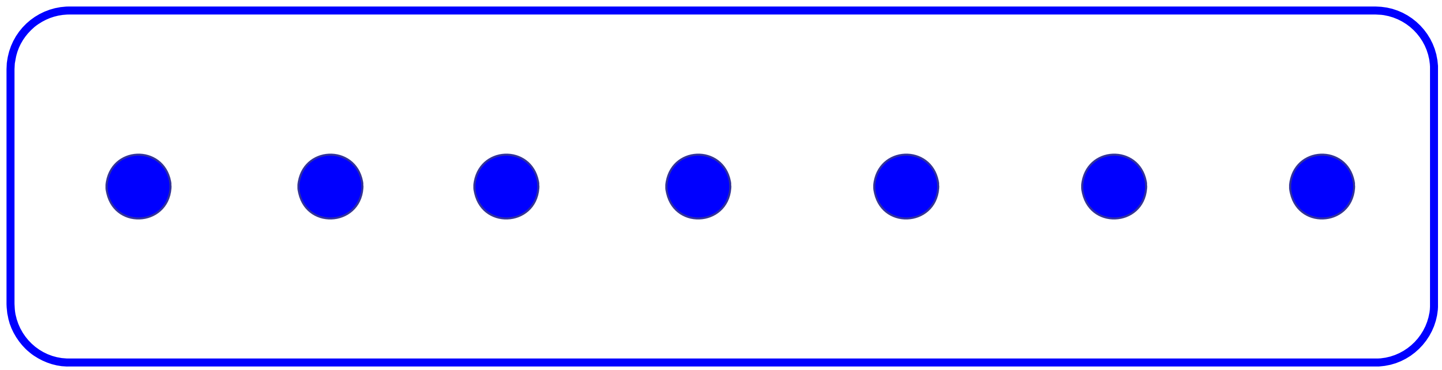
Variants - Syntax (slightly simplified)

- type *name* = C_1 [of ty_1] | . . . | C_n [of ty_n]
- Introduce a type called *name*
- (fun x -> C_i x) : ty_i -> *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

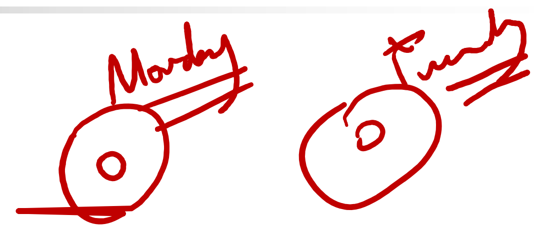


In C and Ocaml they have an order structure; order by order of input

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
```

Problem:

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

- Write function is_weekend : weekday -> bool

let is_weekend day =

match day with
Saturday -> true
| Sunday -> true
| x -> false
_ -> false

Problem:

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

- Write function is_weekend : weekday -> bool

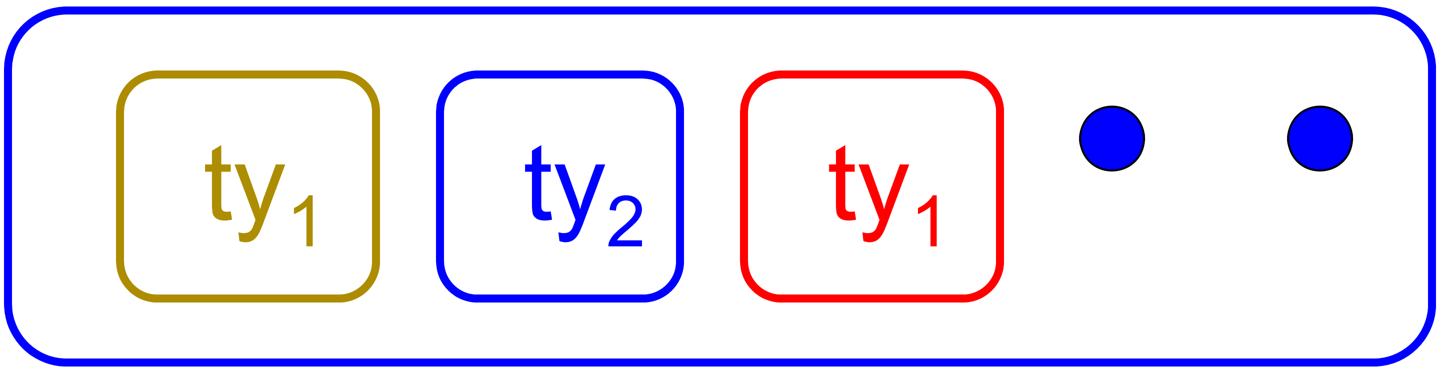
```
let is_weekend day =
    match day with Saturday -> true
     | Sunday -> true
     | _ -> false
```

Example Enumeration Types

```
# type bin_op = IntPlusOp |  IntMinusOp
        |  EqOp | CommaOp | ConsOp

# type mon_op = HdOp | TlOp | FstOp
        | SndOp
```

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;;
type id = DriversLicense of int | SocialSecurity of int | Name of
   string

# 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>
```

Problem

- Create a type to represent the currencies for US, UK, Europe and Japan

Problem

- Create a type to represent the currencies for US, UK, Europe and Japan

```
type currency =
   Dollar of int
 | Pound of int
 | Euro of int
 | Yen of int
```

Example Disjoint Union Type

```
# type const =
    BoolConst of bool
  | IntConst of int
  | FloatConst of float
  | StringConst of string
  | NilConst
  | UnitConst
```

Example Disjoint Union Type

```
# type const = BoolConst of bool
    | IntConst of int | FloatConst of float
    | StringConst of string  | NilConst
    | UnitConst
```

- How to represent 7 as a const?
- Answer: IntConst 7

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 producing 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
```

Functions over option

```
# let result_ok r =
   match r with None -> false
   | Some _ -> true;;
val result_ok : 'a option -> bool = <fun>
# result_ok (first (fun x -> x > 3) [1;3;4;2;5]);;
- : bool = true
# result_ok (first (fun x -> x > 5) [1;3;4;2;5]);;
- : bool = false
```

Problem

- Write a hd and tl on lists that doesn't raise an exception and works at all types of lists.

Problem

- Write a hd and tl on lists that doesn't raise an exception and works at all types of lists.

- let hd list =

  match list with [] -> None

  | (x::xs) -> Some x
- let tl list =

  match list with [] -> None

  | (x::xs) -> Some xs

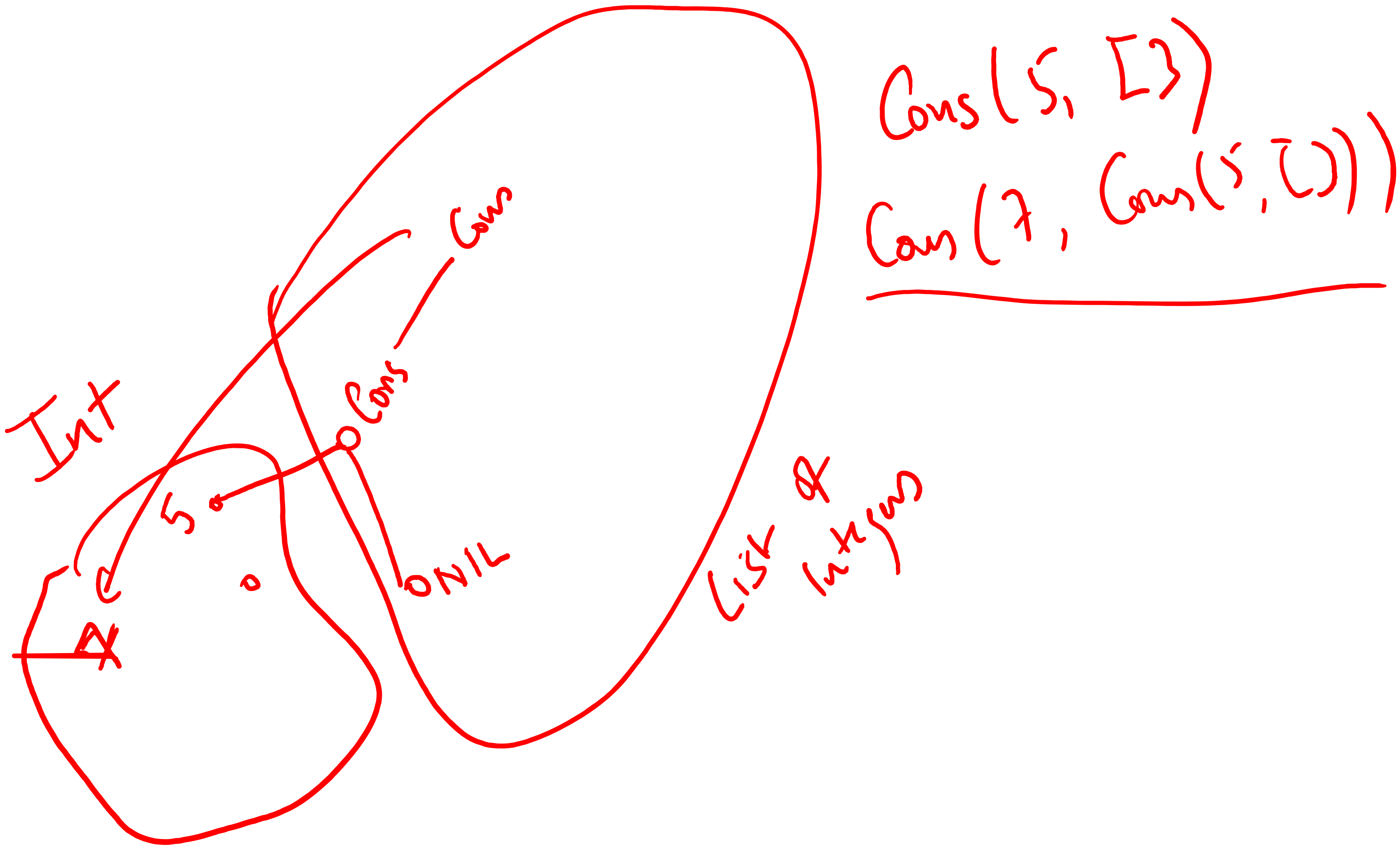
Mapping over Variants

```
# 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

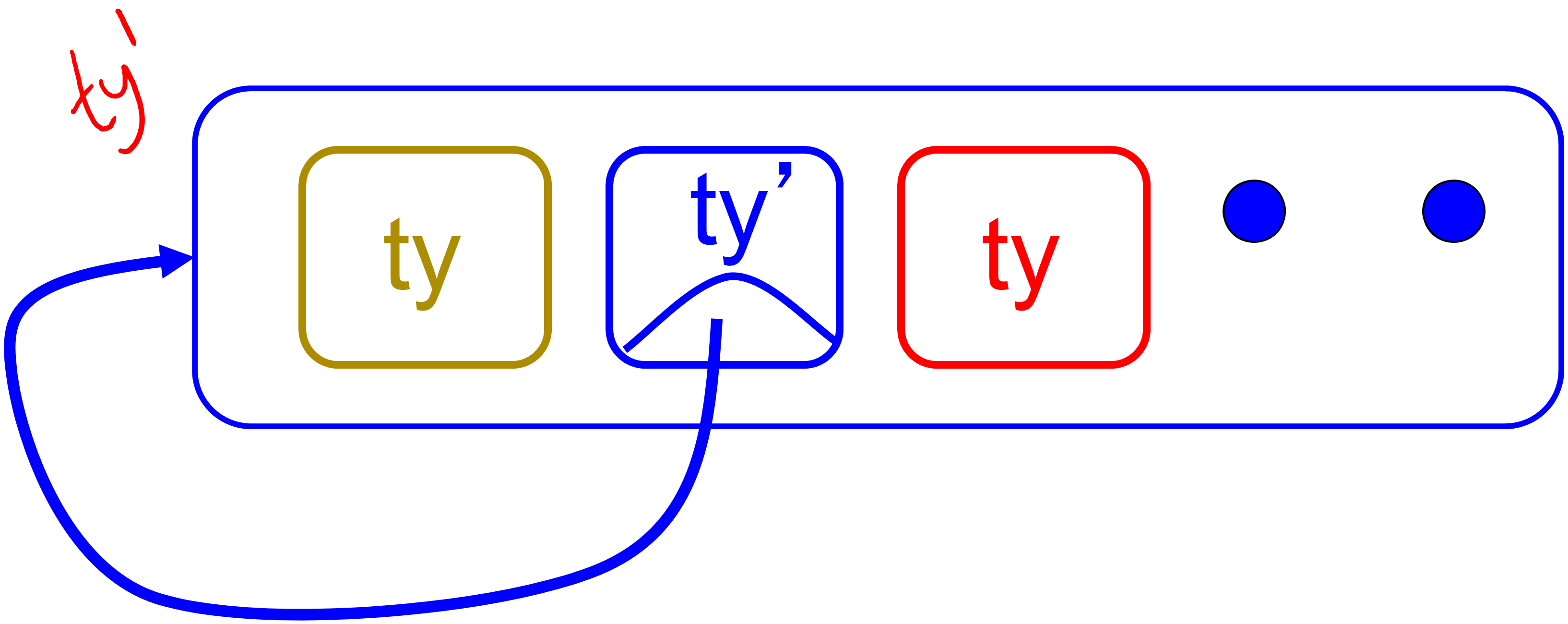
```
# let optionFold someFun noneVal opt =
    match opt with None -> noneVal
    | Some x -> someFun x;;
val optionFold : ('a -> 'b) -> 'b -> 'a option ->
  'b = <fun>
# let optionMap f opt =
    optionFold (fun x -> Some (f x)) None opt;;
val optionMap : ('a -> 'b) -> 'a option -> 'b
  option = <fun>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

Thinking of disjoint union types



Recursive Types

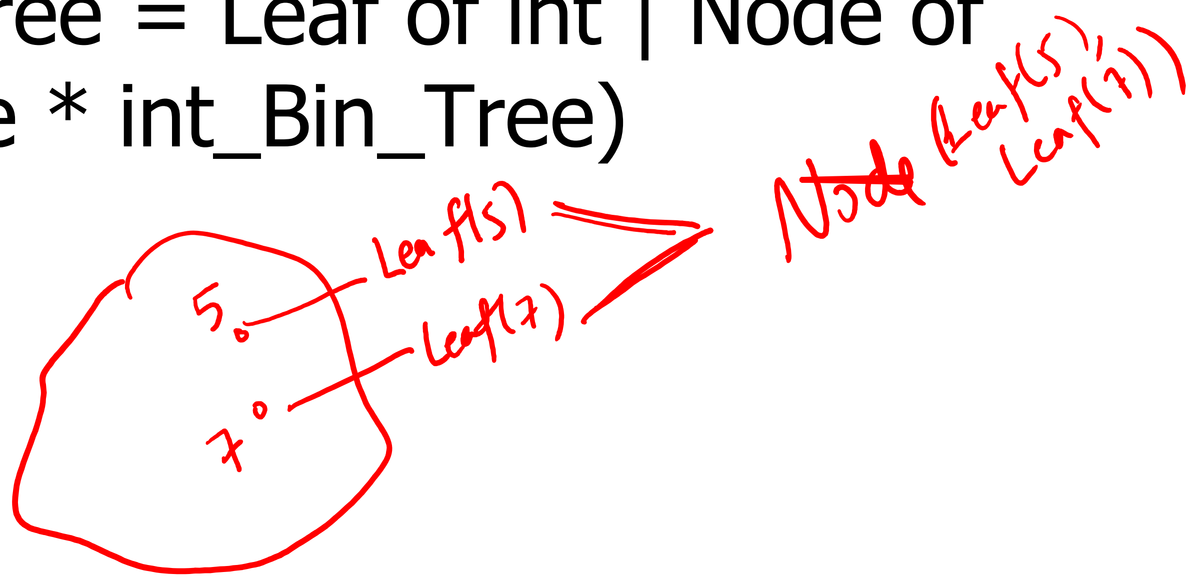
- The type being defined may be a component of itself



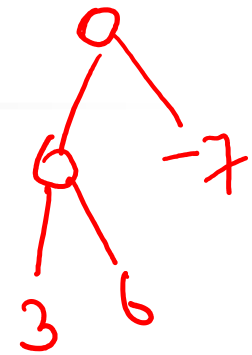
Recursive Data Types

```
# 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)
```



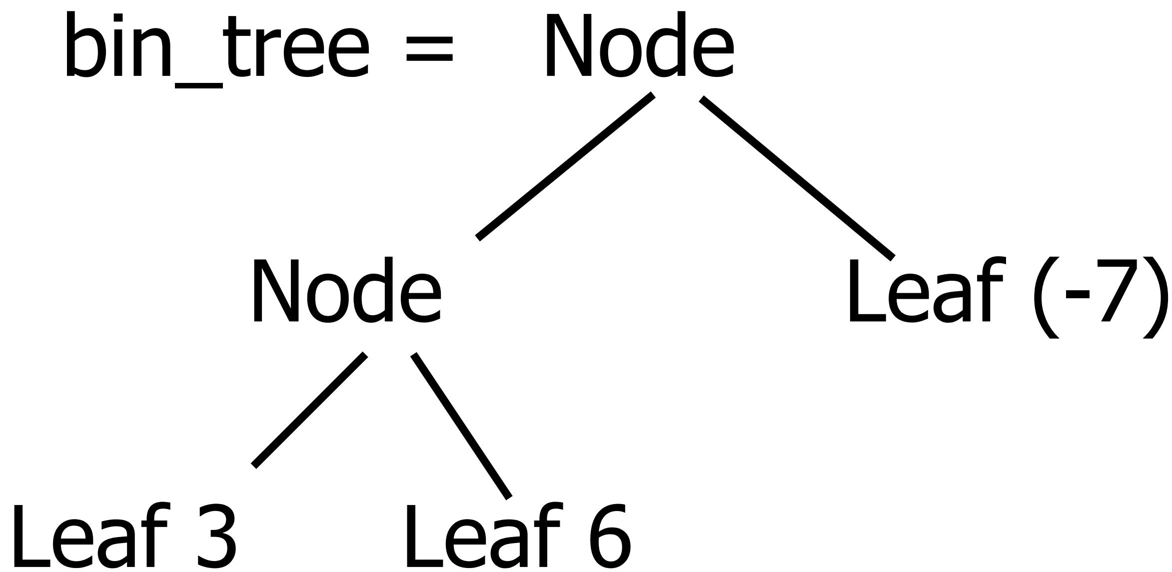
Recursive Data Type Values




```
# 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



Thinking of disjoint union types

Recursive Data Types

```
# 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
```

Thinking of disjoint union types

Symbolic expressions as a recursive data type

Recursive Data Types

type bin_op = IntPlusOp | IntMinusOp
        | EqOp | CommaOp | ConsOp | …
type const = BoolConst of bool | IntConst of int | …
type exp = VarExp of string | ConstExp of const
    | BinOpAppExp of bin_op * exp * exp | …

- How to represent 6 as an exp?

Exp   ConstExp IntConst 6

Recursive Data Types

```
# type bin_op = IntPlusOp |  IntMinusOp
        |  EqOp | CommaOp | ConsOp | …
# type const = BoolConst of bool | IntConst of int |
…
# type exp = VarExp of string | ConstExp of const
  | BinOpAppExp of bin_op * exp * exp  | …
```

- How to represent 6 as an exp?
- Answer: ConstExp (IntConst 6)

Recursive Data Types

type bin_op = IntPlusOp | IntMinusOp
 | EqOp | CommaOp | ConsOp | …

type const = BoolConst of bool | IntConst of int | …

type exp = VarExp of string | ConstExp of const
 | BinOpAppExp of bin_op * exp * exp | …

- How to represent (6, 3) as an exp?

Recursive Data Types

```
# type bin_op = IntPlusOp |  IntMinusOp
        |  EqOp | CommaOp | ConsOp | …
# type const = BoolConst of bool | IntConst of int |
…
# type exp = VarExp of string | ConstExp of const
   | BinOpAppExp of bin_op * exp * exp  | …
```

- How to represent (6, 3) as an exp?
- BinOpAppExp (CommaOp, ConstExp (IntConst 6), ConstExp (IntConst 3))

Recursive Data Types

```
# type bin_op = IntPlusOp |  IntMinusOp
        |  EqOp | CommaOp | ConsOp | …
# type const = BoolConst of bool | IntConst of int |
…
# type exp = VarExp of string | ConstExp of const
   | BinOpAppExp of bin_op * exp * exp  | …
```

- How to represent [(6, 3)] as an exp?
- BinOpAppExp (ConsOp, BinOpAppExp (CommaOp, ConstExp (IntConst 6), ConstExp (IntConst 3)), ConstExp NilConst))));;

Recursive Data Types

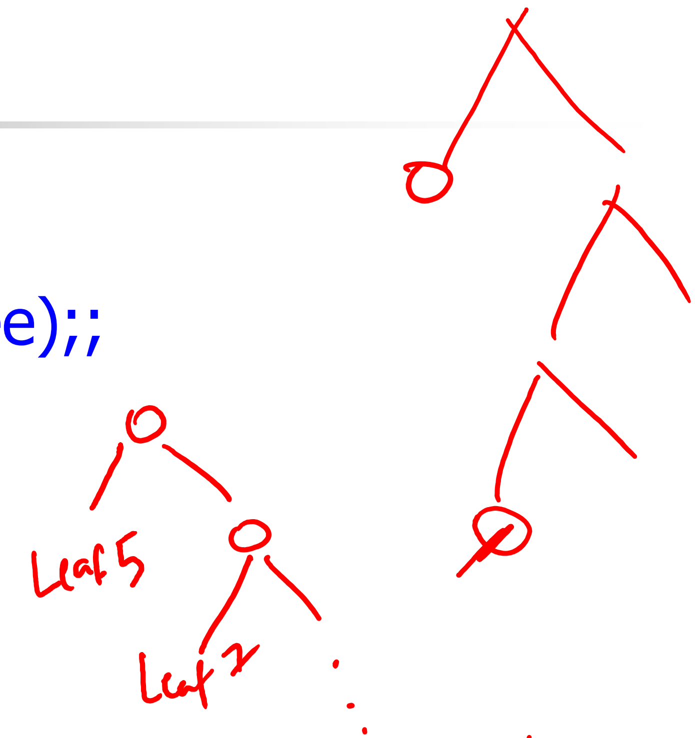
```
# type int_Bin_Tree =
 Leaf of int | Node of (int_Bin_Tree * int_Bin_Tree);;

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

# 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 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 (lt, rt) -> first_leaf_value lt



0

type Mylist = NilList | MyConsOp int MyList

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
```

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 tree

let rec sum_tree t =

```
match t with Leaf n -> n
   | Node (lt, rt) ->
       (sum_tree lt) + (sum_tree rt)
```

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 tree

```
let rec sum_tree t =
    match t with Leaf n -> n
    | Node(t1,t2) -> sum_tree t1 + sum_tree t2
```

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 occurrences of variables in an exp?

let ~~rec~~ count e = match e with

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
```

- Count the number of occurrences 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) ->
```

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
```

- Count the number of occurrences 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
```

Mapping over Recursive Types

```
# let rec ibtreeMap f tree =
   match tree with (Leaf n) ->
    | Node (left_tree, right_tree) ->
```

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))

Node (Node (Leaf 7, Leaf 10), Leaf (-3))

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

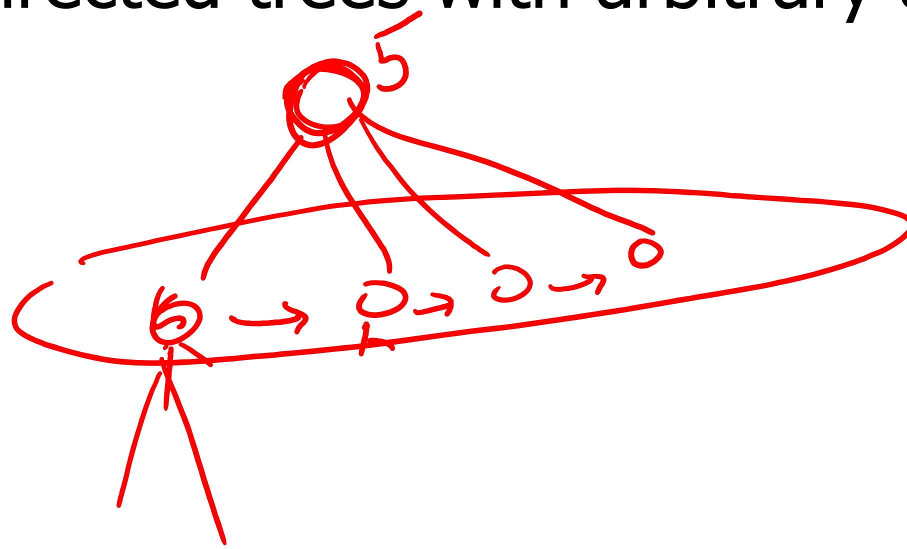
```
# let tree_sum =
    ibtreeFoldRight (fun x -> x) (+);;
val tree_sum : int_Bin_Tree -> int = <fun>
# tree_sum bin_tree;;
- : int = 2
```

Mutually Recursive Types

Type T1's definition has type T2

Type T2's definition has type T1

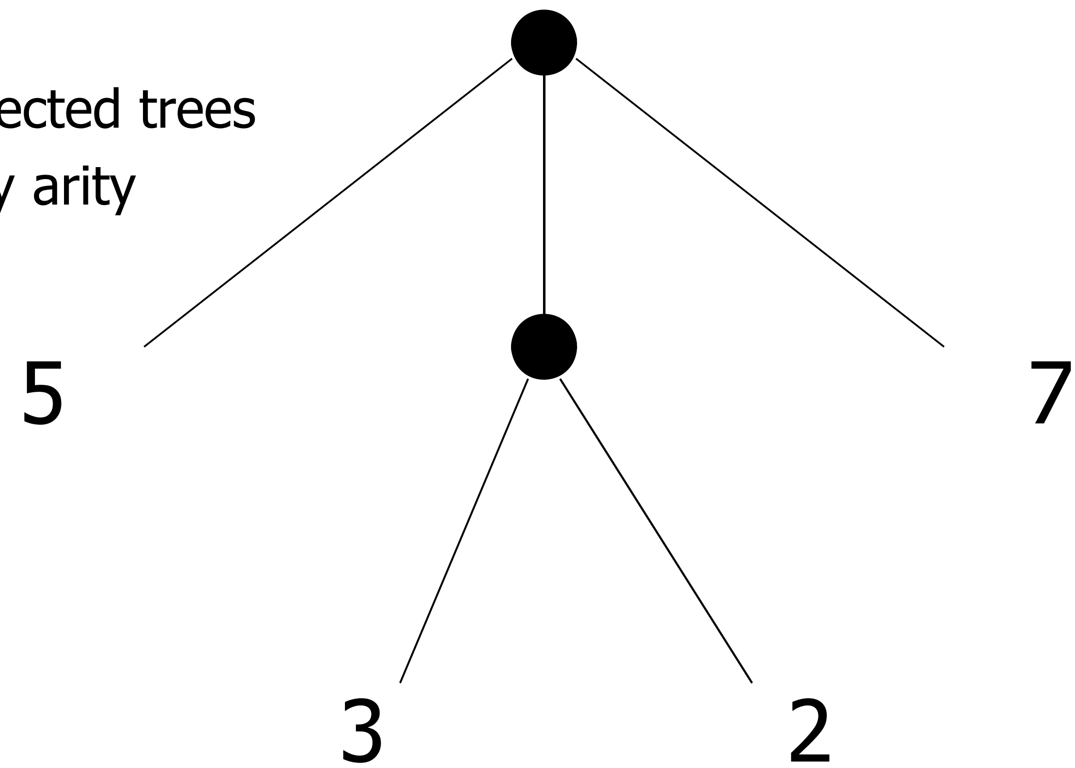
Example: directed trees with arbitrary arity



Mutually Recursive Types - Values

Type T1's definition has type T2
Type T2's definition has type T1

Example: directed trees
with arbitrary arity



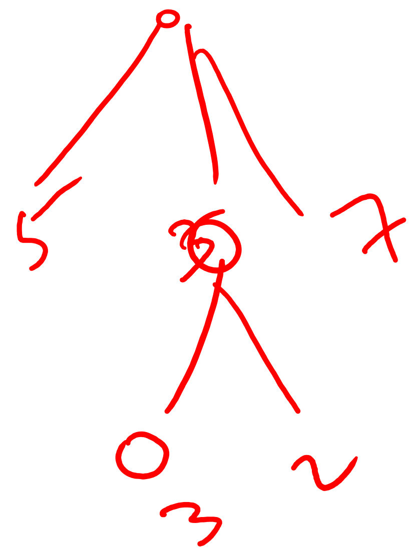
Mutually Recursive Types

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

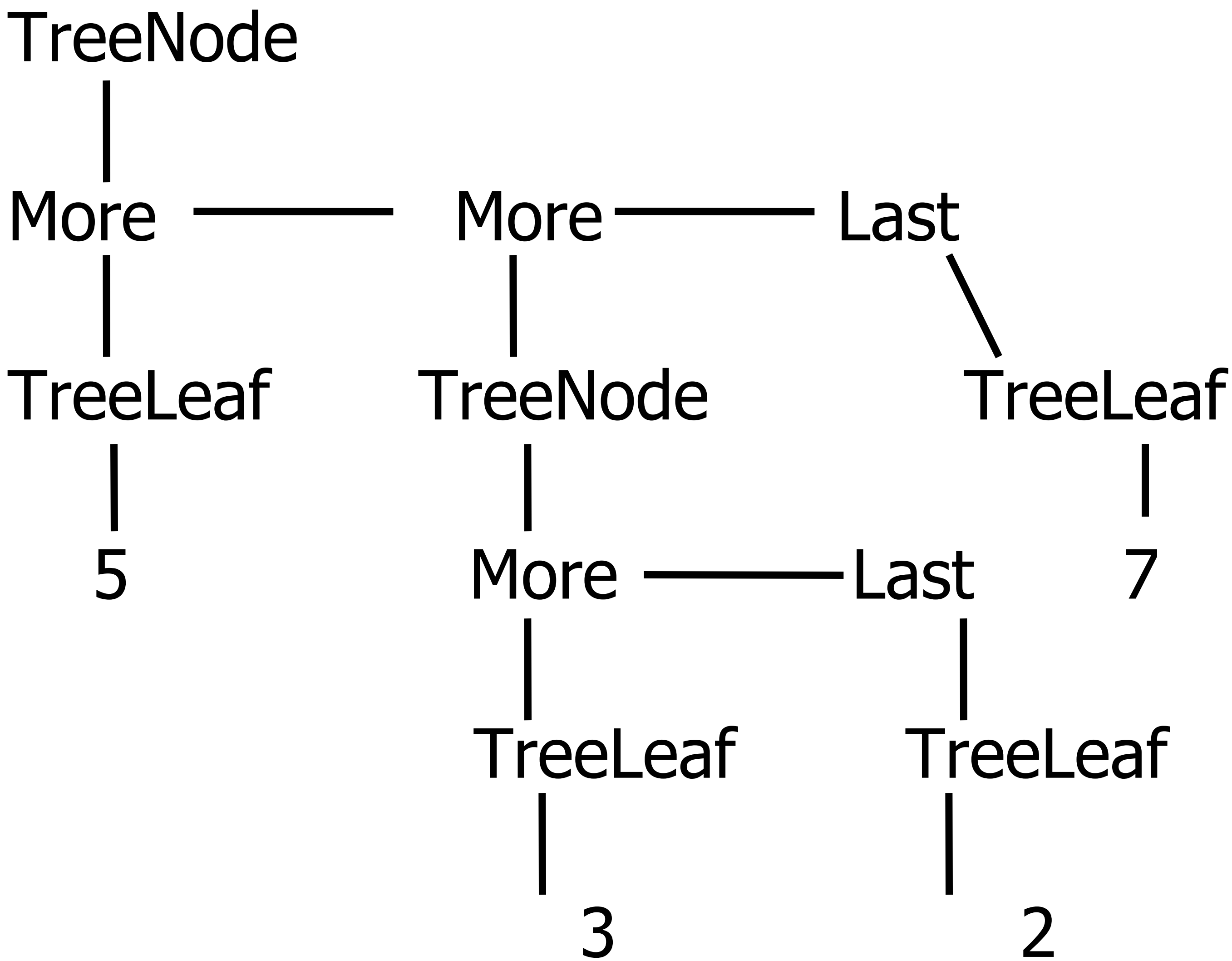
```
type 'a tree = TreeLeaf of 'a | TreeNode of 'a treeList
and 'a treeList = Last of 'a tree | More of ('a tree * 'a treeList)
```

Mutually Recursive Types - Values

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

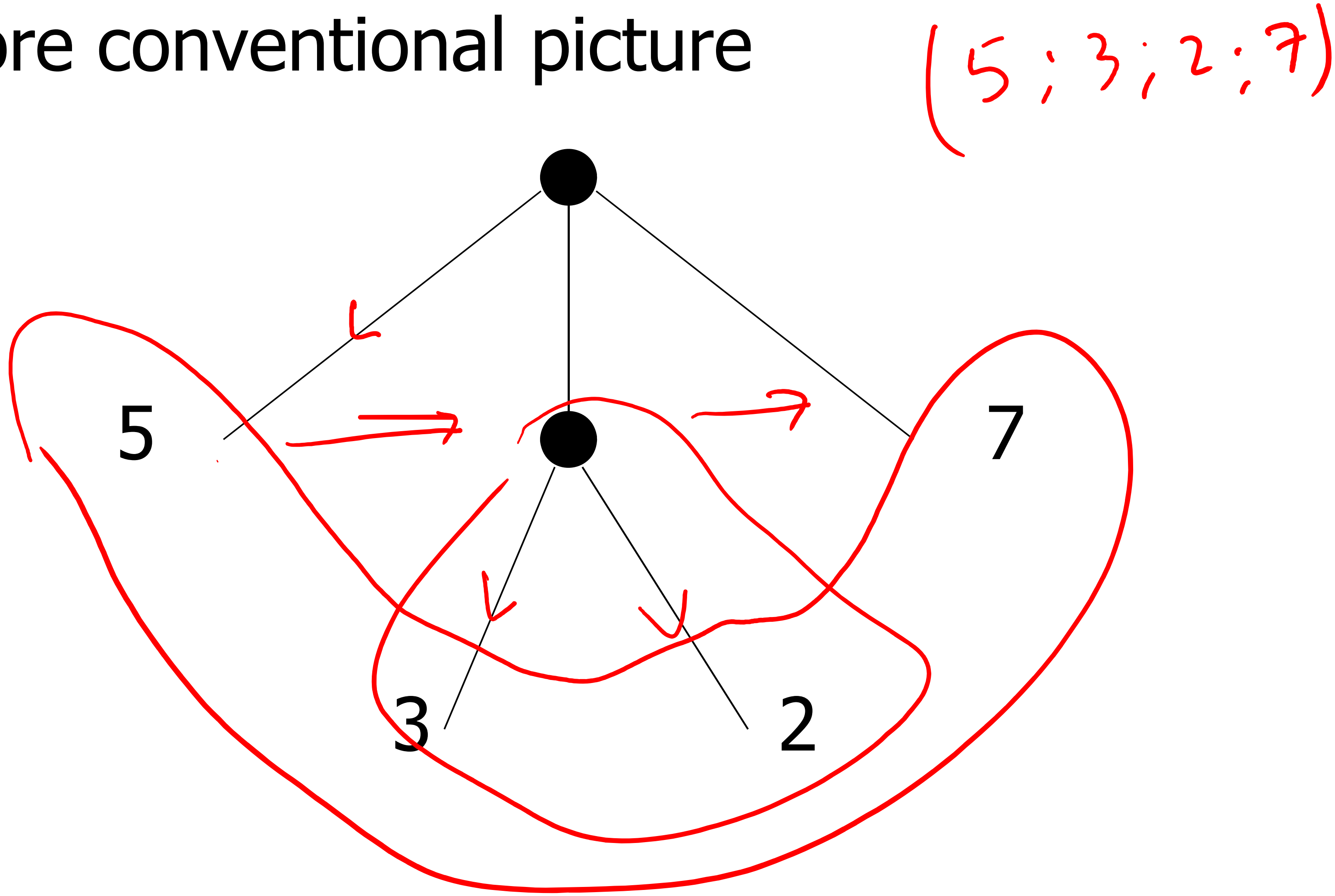


Mutually Recursive Types - Values



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]
```

Problem

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

Define tree_size

Problem

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

Define tree_size

```
let rec tree_size t =
     match t with TreeLeaf _ ->
     | TreeNode ts ->
```

Problem

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

Define tree_size

```
let rec tree_size t =
    match t with TreeLeaf _ -> 1
    | TreeNode ts -> treeList_size  ts
```

Problem

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

Define tree_size and treeList_size

```
let rec tree_size t =
      match t with TreeLeaf _ -> 1
      | TreeNode ts -> treeList_size  ts
and treeList_size ts =
```

Problem

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

Define tree_size and treeList_size

```
let rec tree_size t =
      match t with TreeLeaf _ -> 1
      | TreeNode ts -> treeList_size  ts
and treeList_size ts =
      match ts with Last t ->
      | More t ts’ ->
```

Problem

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

Define tree_size and treeList_size

```
let rec tree_size t =
    match t with TreeLeaf _ -> 1
    | TreeNode ts -> treeList_size  ts
and treeList_size ts =
    match ts with Last t -> tree_size t
    | More t ts' -> tree_size t + treeList_size ts'
```

Problem

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

Define tree_size and treeList_size

```
let rec tree_size t =
    match t with TreeLeaf _ -> 1
    | TreeNode ts -> treeList_size  ts
and treeList_size ts =
    match ts with Last t -> tree_size t
    | More t ts' -> tree_size t + treeList_size ts'
```

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)
```

Mindblowing!

What does this mean?

What's the base case?!

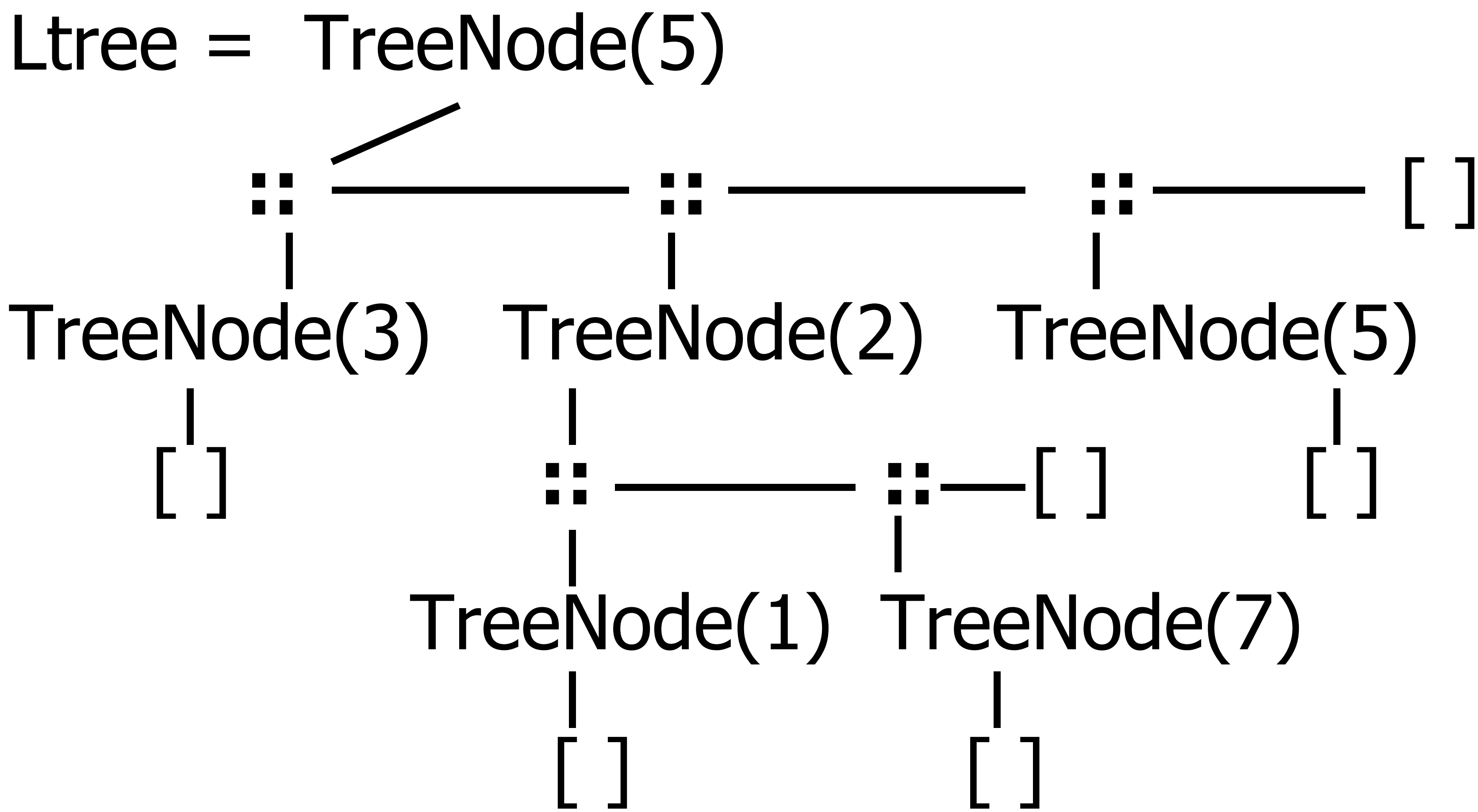
Nested Recursive Type Values

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

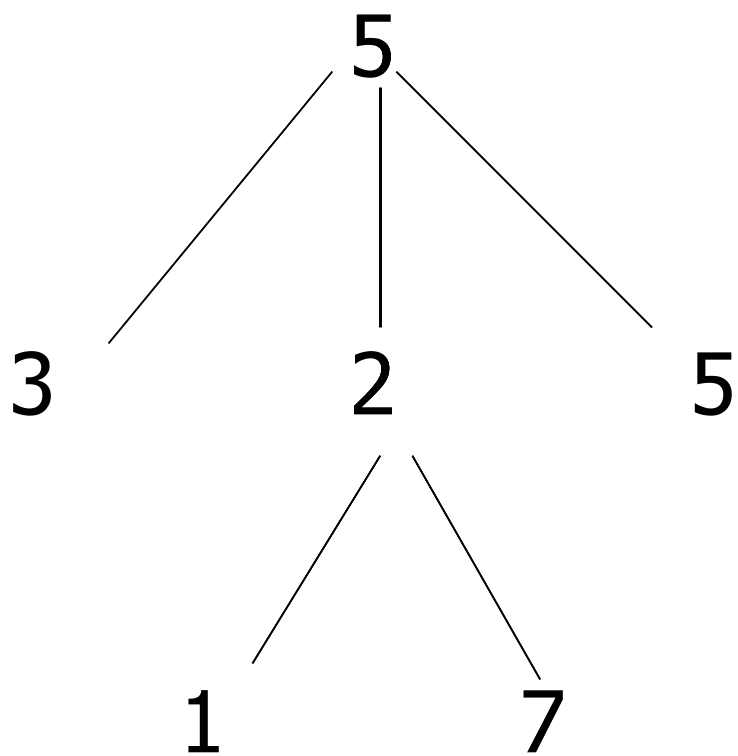
Nested Recursive Type Values

```
val ltree : int labeled_tree =
  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

Infinite Recursive Values

```
# let rec ones = 1::ones;;
val ones : int list =
  [1; 1; 1; 1; ...]
# match ones with x::_ -> x;;
Characters 0-25:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
  match ones with x::_ -> x;;
  ^^^^^^^^^^^^^^^^^^^^^^^^^
- : int = 1
```

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 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)
```

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
```

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 "Guieseppe Martin" (523,04,6712)
  student;;
- : person = {name = "Guieseppe Martin"; ss
  = (523, 4, 6712); age = 22}
```