Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

http://www.cs.uiuc.edu/class/cs421/

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

CPS Transformation

- Step 1: Add continuation argument to any function definition:
  - let f arg = e ⇒ let f arg k = e
  - Idea: Every function takes an extra parameter saying where the result goes
- Step 2: A simple expression in tail position should be passed to a continuation instead of returned:
  - return a ⇒ k a
  - Assuming a is a constant or variable.
  - "Simple" = "No available function calls."

CPS Transformation

- Step 3: Pass the current continuation to every function call in tail position
  - return f arg_1 … arg_n ⇒ f arg_1 … arg_n k
  - The function "isn't going to return," so we need to tell it where to put the result.

CPS Transformation

- Step 4: Each function call not in tail position needs to be built into a new continuation (containing the old continuation as appropriate)
  - return op (f arg) ⇒ f arg (fun r -> k(op r))
  - op represents a primitive operation

  - return f(g arg) ⇒ g arg (fun r-> f r k)

Example

Before:

```
let rec add_list lst =
match lst with
  [ ] -> 0
| 0 :: xs -> add_list xs
| x :: xs -> (+) x
   (add_list xs);;
```

After:

```
let rec add_listk lst k =
                        (* rule 1 *)
match lst with
| [ ] -> k 0 (* rule 2 *)
| 0 :: xs -> add_listk xs k
                           (* rule 3 *)
| x :: xs -> add_listk xs
           (fun r -> k ((+) x r));;
                     (* rule 4 *)
```

Other Uses for Continuations

- CPS designed to preserve order of evaluation
- Continuations used to express order of evaluation
- Can be used to change order of evaluation
- Implements:
  - Exceptions and exception handling
  - Co-routines
  - (pseudo) threads

Exceptions - Example

```
# exception Zero;;
exception Zero
# let rec list_mult_aux list =
    match list with [ ] -> 1
    | x :: xs ->
     if x = 0 then raise Zero
                   else x * list_mult_aux xs;;
val list_mult_aux : int list -> int = <fun>
```

Exceptions - Example

```
# let list_mult list =
    try list_mult_aux list with Zero -> 0;;
val list_mult : int list -> int = <fun>
# list_mult [3;4;2];;
- : int = 24
# list_mult [7;4;0];;
- : int = 0
# list_mult_aux [7;4;0];;
Exception: Zero.
```

Exceptions

- When an exception is raised
  - The current computation is aborted
  - Control is "thrown" back up the call stack until a matching handler is found
  - All the intermediate calls waiting for a return value are thrown away

Implementing Exceptions

```
# let multkp m n k =
   let r = m * n in
    (print_string "product result: ";
     print_int r; print_string "\n";
     k r);;
val multkp : int -> int -> (int -> 'a) -> 'a
  = <fun>
```

Implementing Exceptions

```
# let rec list_multk_aux list k kexcp =
    match list with [ ] -> k 1
    | x :: xs -> if x = 0 then  kexcp  0
      else list_multk_aux xs
              (fun r -> multkp x r k) kexcp;;
val list_multk_aux : int list -> (int -> 'a) -> (int -> 'a)
   -> 'a = <fun>
# let rec list_multk list k = list_multk_aux list  k  k;;
val list_multk : int list -> (int -> 'a) -> 'a = <fun>
```

Implementing Exceptions

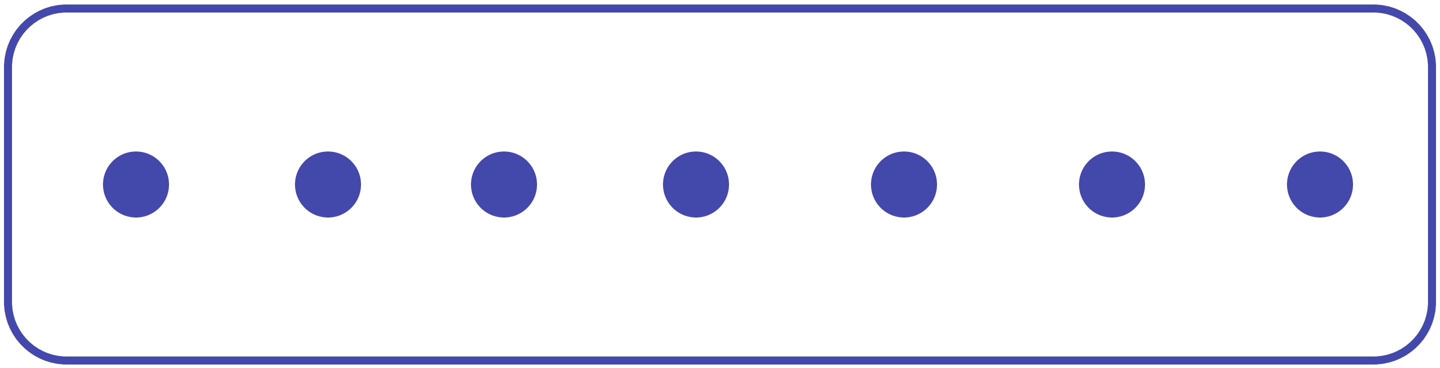
```
# list_multk [3;4;2] report;;
product result: 2
product result: 8
product result: 24
24
- : unit = ()
# list_multk [7;4;0] report;;
0
- : unit = ()
```

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_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



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

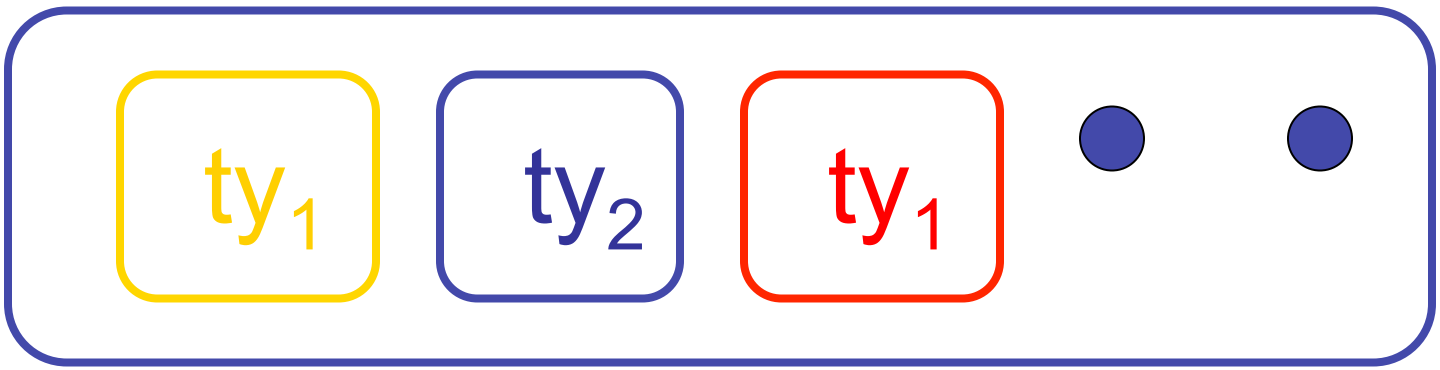
```
# let rec days_later n day =
    match n with 0 -> day
    | _ -> if n > 0
         then day_after (days_later (n - 1) day)
        else days_later (n + 7) day;;
val days_later : int -> weekday -> weekday
  = <fun>
```

Functions over Enumerations

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

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>
```

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

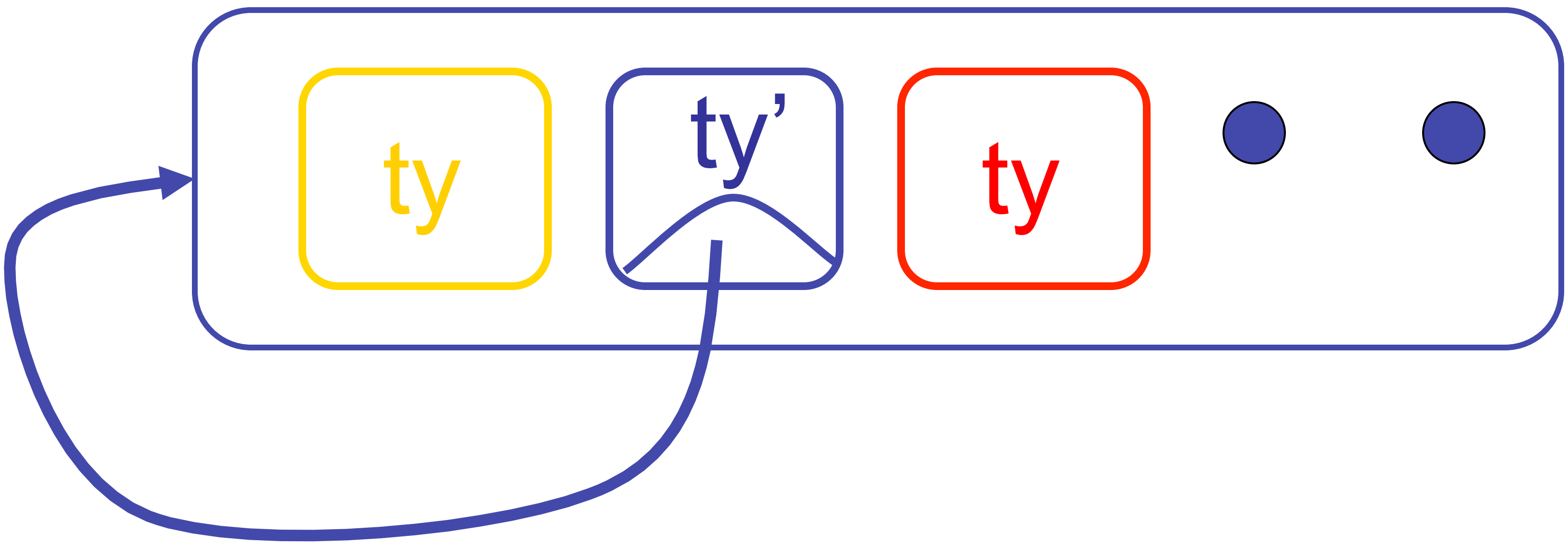
```
# let optionMap f opt =
    match opt with None -> None
    | Some x -> Some (f x);;
val optionMap : ('a -> 'b) -> 'a option -> 'b
  option = <fun>
# optionMap
  (fun x -> x - 2)
  (first (fun x -> x > 3) [1;3;4;2;5]);;
- : int option = Some 2
```

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 Data Types

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

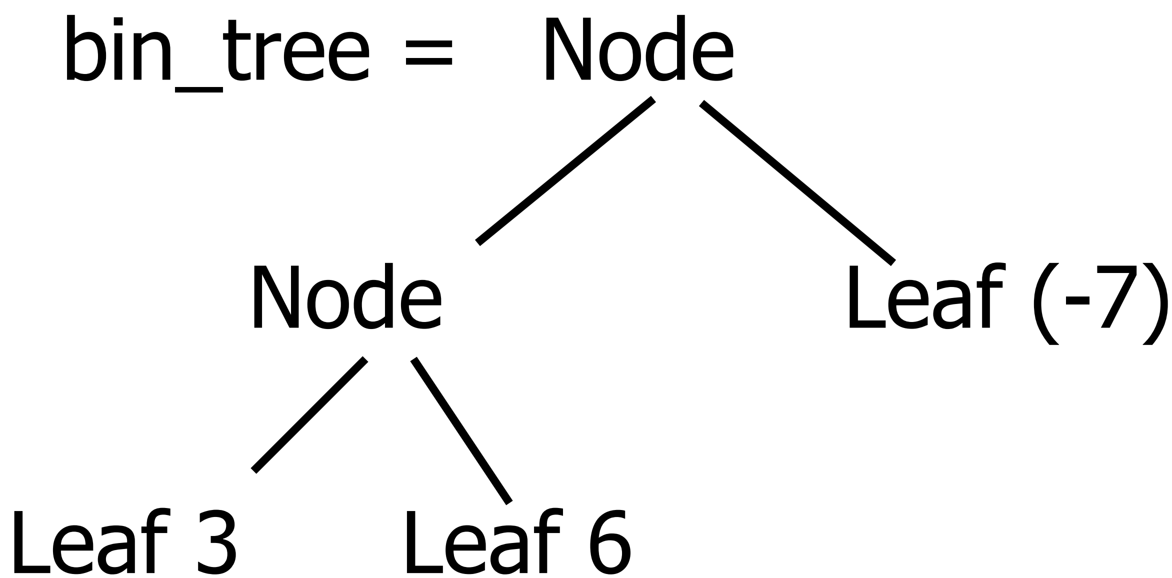
```
type int_Bin_Tree = Leaf of int | Node of
  (int_Bin_Tree * int_Bin_Tree)
```

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



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
```

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)

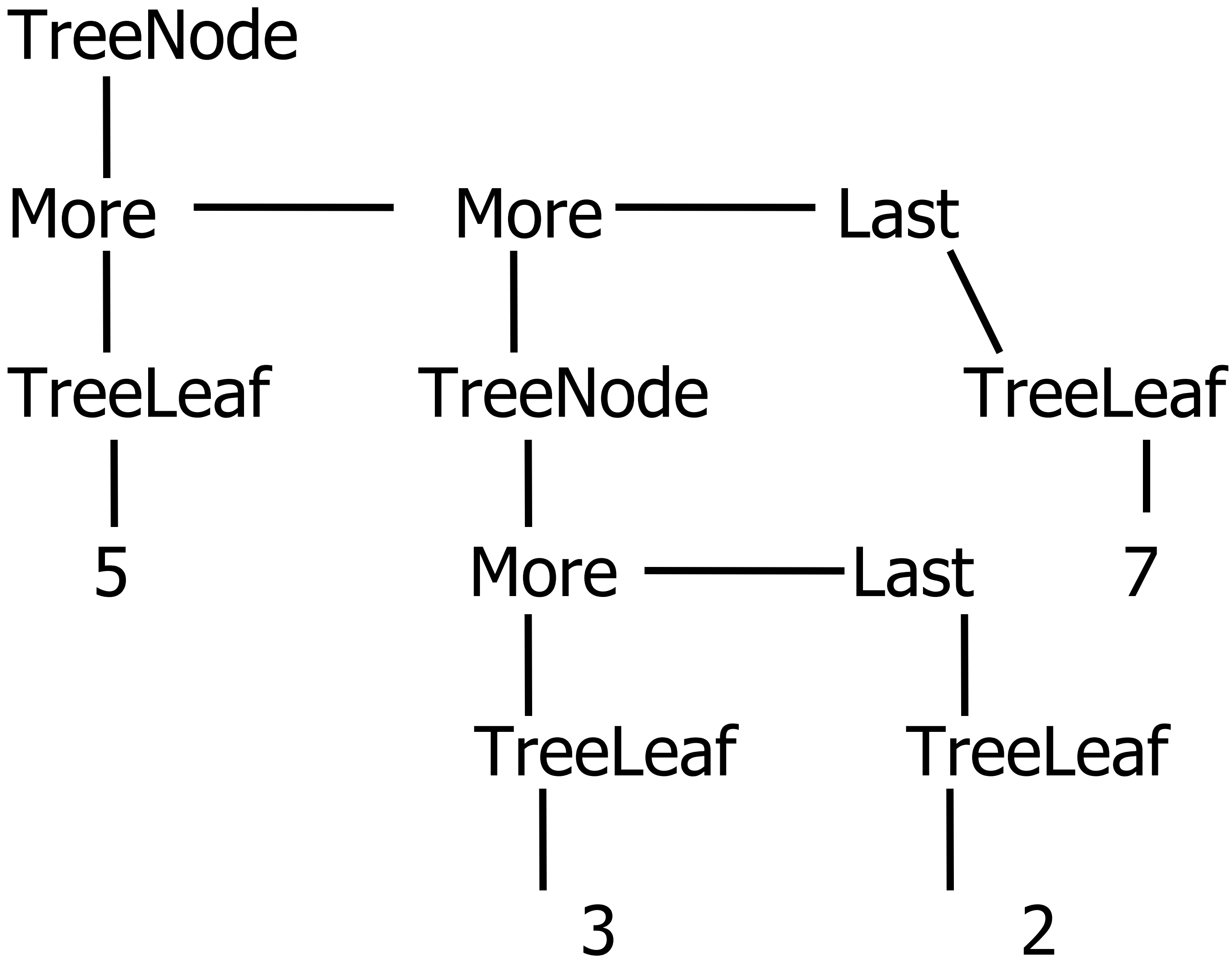
Mutually Recursive Types - Values

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

Mutually Recursive Types - Values

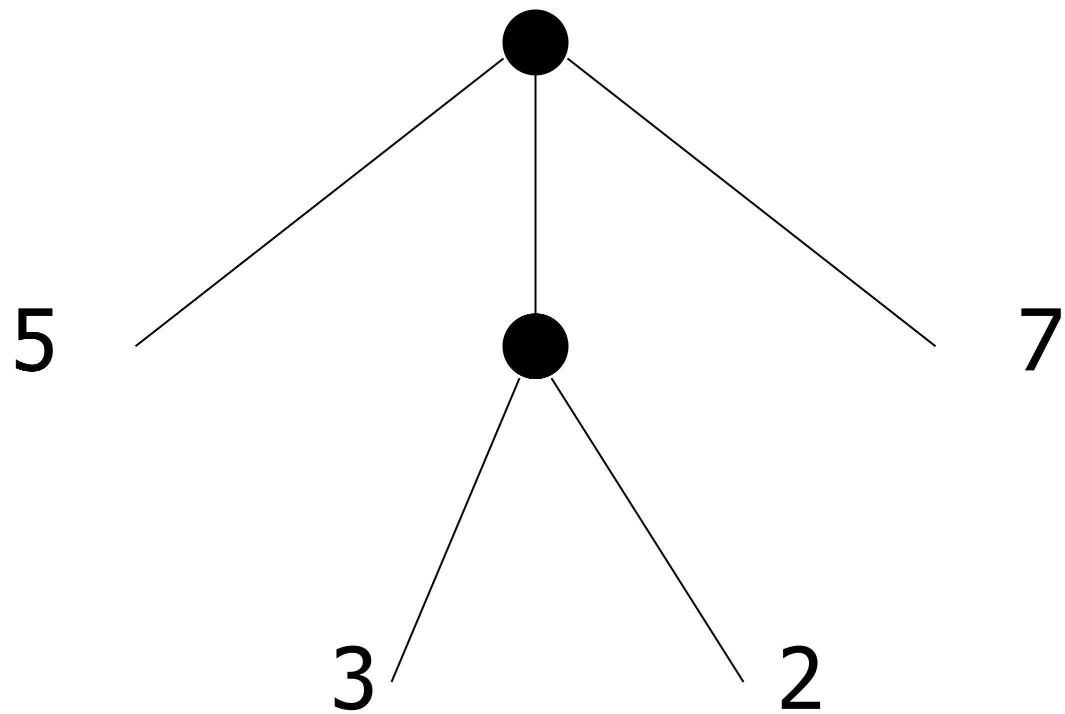
```
val tree : int tree =
 TreeNode
  (More
    (TreeLeaf 5,
     More
      (TreeNode (More (TreeLeaf 3, Last
  (TreeLeaf 2))), Last (TreeLeaf 7))))
```

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)
```

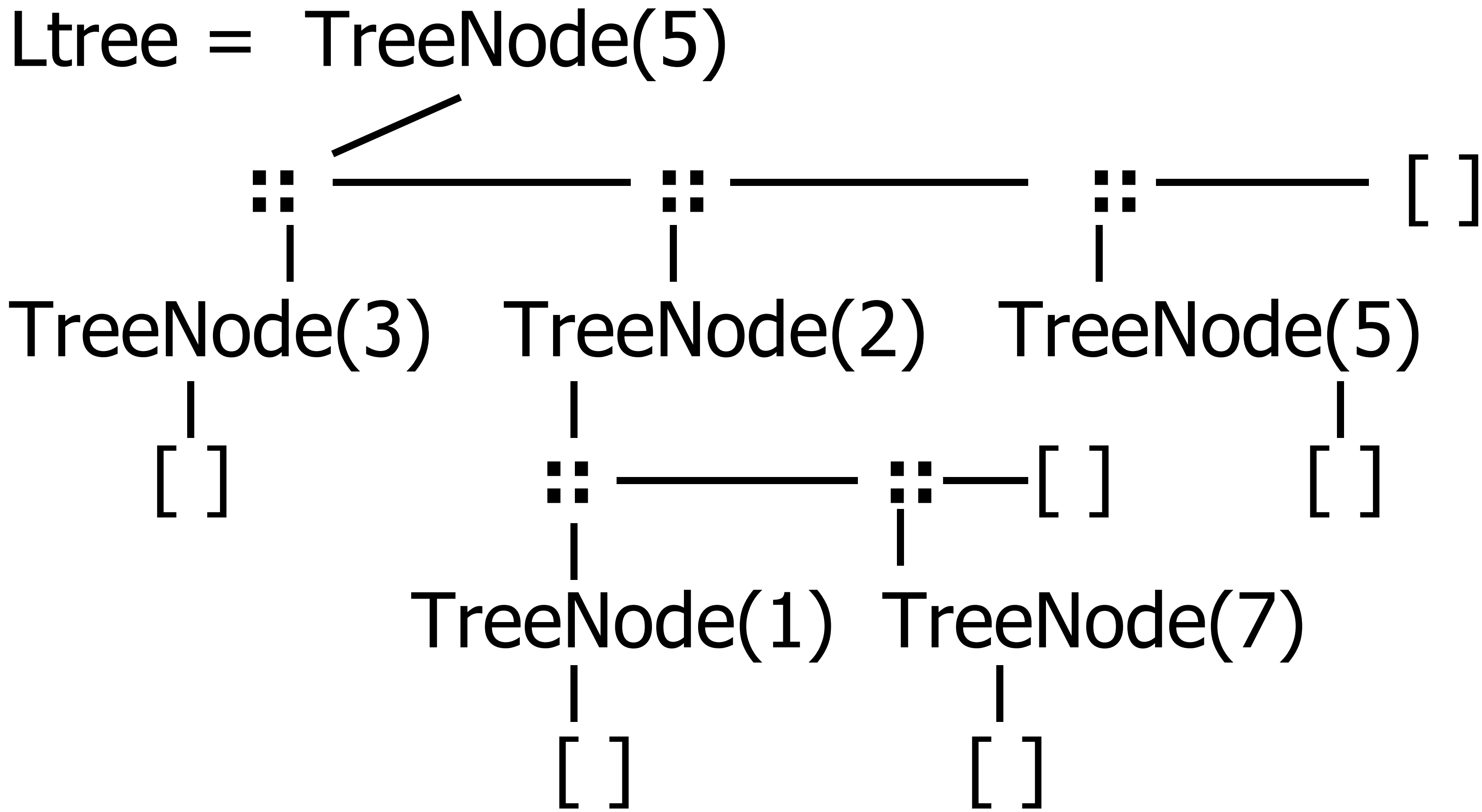
Nested Recursive Type Values

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

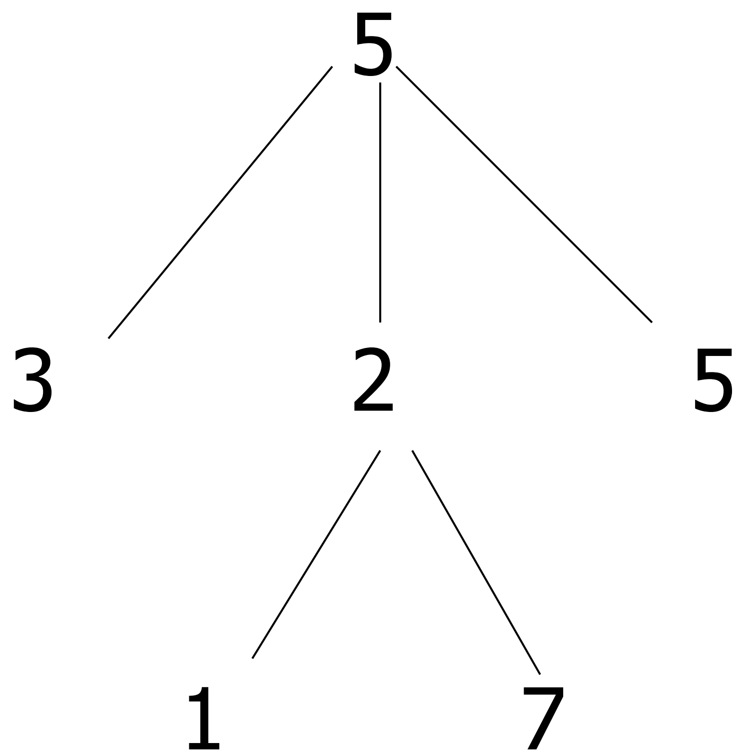
Nested Recursive Type Values

```
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];;
```

Infinite Recursive Values

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