Recursive Types

- The type being defined may be a component of itself

```
  ty ty' ty
```

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
```

How to represent 6 as an `exp`?

- Answer: `ConstExp (IntConst 6)`

```
# 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))

Your turn now

Try Problem 3 on MP3
Mutually Recursive Types

```plaintext
# 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

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

Mutually Recursive Types - Values

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

Mutually Recursive Functions

```plaintext
# 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);
```

```plaintext
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 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);
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 Type Values

let ltree =
  TreeNode(5,
    [TreeNode (3, []);
     TreeNode (2, [TreeNode (1, []);
                   TreeNode (7, [])]);
     TreeNode (5, [])]);
val ltree : int labeled_tree =
  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

Ltree =  TreeNode(5)
          ::                ::                 ::      ... 
                   TreeNode(1)  TreeNode(7) 
                       [ ]              [ ]

Ltree =  TreeNode(5)
          ::                ::                 ::      ... 
       [ ]
       TreeNode(3)  TreeNode(2)  TreeNode(5)
       ::                ::                 ::      ... 
       [ ]
       TreeNode(1)  TreeNode(7) 
       [ ]              [ ]
Nested Recursive Type Values

Mutually Recursive Functions

```ocaml
# 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

```ocaml
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]
```

Infinite Recursive Values

```ocaml
# let rec ones = 1::ones;;
val ones : int list = [1; 1; 1; 1; ...]
# match ones with x::_ -> x;;
- : int = 1
```

Infinite Recursive Values

```ocaml
# 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

```ocaml
# 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

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