Recursive Data Types

```ocaml
# 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)

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

```ocaml
# 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?

```ocaml
BinOpAppExp (ConsOp, BinOpAppExp (CommaOp, ConstExp (IntConst 6), ConstExp (IntConst 3)), ConstExp NilConst));;
```

Problem

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

Write `sum_tree : int_Bin_Tree -> int`

```ocaml
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

```ocaml
# 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?

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

```ocaml
# 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>
```

```
ibtreeMap ((+) 2) bin_tree;;
#: int_Bin_Tree = Node (Node (Leaf 5, Leaf 8), Leaf (-5))
```

Folding over Recursive Types

```ocaml
# 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>
```

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

```
# tree_sum bin_tree;;
#: int = 2
```

Mutually Recursive Types

```ocaml
# 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)
```

```
# 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

TreeNode
  More
    More
      Last
        TreeLeaf
          5
          More
            Last
              7
              TreeLeaf
                3
                TreeLeaf
                  2

A more conventional picture

# fringe tree;;
- : int list = [5; 3; 2; 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);;

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

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 ->

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 ->
    | 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'
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

Ltree =  TreeNode(5)  
        |   |  |
        |   |  |
        |   |  |
        |   |  |
        |   |  |
        |   |  |

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

```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]
```

Nested recursive types lead to mutually recursive functions

Extra Material

Infinite Recursive Values

```ocaml
# 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:

```
```
```
- : 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
# 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; };

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
```

Record Field Access

```ocaml
# let soc_sec = teacher.ss;;

val soc_sec : int * int * int = (119, 73, 6244)
```

Record Values

```ocaml
# let student = {ss=(325,40,1276); name="Joseph Martins"; age=22};;

val student : person = {name = "Joseph Martins"; ss = (325, 40, 1276); age = 22}
```

Record Pattern Matching

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

val els : string = "Elsa L. Gunter"
val age : int = 102
val s3 : int = 6244
```

Record Field Access

```ocaml
# let soc_sec = teacher.ss;;

val soc_sec : int * int * int = (119, 73, 6244)
```
# 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}

# 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}