Example

Before:
let rec mem (y, lst) =
match lst with
  | [] -> false
  | x :: xs ->
    if (x = y)
    then true
    else mem(y, xs)

After:
let rec memk (y, lst) k =
match lst with
  | [] -> false
  | x :: xs ->
    eqk(x, y)
    (fun b ->
      if b
      then k true (* rule 2 *)
      else memk(y, xs) (* rule 3 *)
    )

Example

Before:
let rec mem (y, lst) =
match lst with
  | [] -> false
  | x :: xs ->
    if (x = y)
    then true
    else mem(y, xs)

After:
let rec memk (y, lst) k =
match lst with
  | [] -> false
  | x :: xs ->
    eqk(x, y)
    (fun b ->
      if b
      then k true (* rule 2 *)
      else memk(y, xs) (* rule 3 *)
    )
Example

Before:
let rec mem (y,lst) =
match lst with
  [] -> false
| x :: xs ->
  if (x = y) then true
  else mem(y,xs);;

After:
let rec memk (y,lst) k =
(* rule 1 *)
match lst with
  | [] -> k false (* rule 2 *)
  | x :: xs ->
    eqk (x, y)
    (fun b -> if b (* rule 4 *)
     then k true (* rule 2 *)
     else memk(y, xs) k (* rule 3 *)

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

Variants - Syntax (slightly simplified)

- type name = C_i [of ty_i] | . . . | 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

\[
\begin{array}{cccccccc}
  1 & 2 & 3 & 4 & 5 & 6 & 7 & 8
\end{array}
\]

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

```ocaml
# 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>
```

```ocaml
# 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>
```

```ocaml
# days_later 2 Tuesday;;
- : weekday = Thursday

# days_later (-1) Wednesday;;
- : weekday = Tuesday

# days_later (-4) Monday;;
- : weekday = Thursday
```

```ocaml
# 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
```

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

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

Example Enumeration Types
Disjoint Union Types
- Disjoint union of types, with some possibly occurring more than once
- We can also add in some new singleton elements

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

Example Disjoint Union Type
- 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.

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

Functions producing option

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

```ocaml
# 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

```ocaml
# 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.

```ocaml
# let hd list =  
  match list with [] -> None  
  | (x::xs) -> Some x
val hd : 'a list -> 'a option = <fun>
```

```ocaml
# let tl list =  
  match list with [] -> None  
  | (x::xs) -> Some xs
val tl : 'a list -> 'a option = <fun>
```

Mapping over Variants

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

```ocaml
# optionMap (fun x -> x - 2)  
  (first (fun x -> x > 3) [1;3;4;2;5]);;
- : int option = Some 2
```
Folding over Variants

```ocaml
# 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

```
let rec ty ty' ty' =
```

Recursive Data Types

```ocaml
# 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

```ocaml
# 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

```
bin_tree =   Node
    Node               Leaf (-7)
  Leaf 3      Leaf 6
```

Recursive Data Types

```ocaml
# 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
```
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)

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

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

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

Write sum_tree : int_Bin_Tree -> int

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

  val varCnt : exp -> int
```

```
Your turn now
Try Problem 3 on MP5
```
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>
```

```ocaml
# 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>
```

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

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)
```

```ocaml
# 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

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 ->

and treeList_size ts =
  match ts with Last t ->
  | More t ts' ->

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' -> tree_size t + treeList_size ts'
Nested Recursive Types

```ocaml
# 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

```ocaml
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

```
Ltree =  TreeNode(5)
        ::                ::                 ::
        [ ]
    TreeNode(3)   TreeNode(2)   TreeNode(5)
        [ ]             ::             ::
        [ ]        [ ]
    TreeNode(1)  TreeNode(7)
        [ ]              [ ]
```

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

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

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

```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)
```

New Records from Old

```ocaml
# 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}
```