

CMSC 330: Organization of Programming Languages

OCaml Data Types

CMSC330 Fall 2025

Review: Fold

```
let rec fold_left f a l =  
  match l with  
  [ ] -> a  
  | h::t -> fold_left f (f a h) t
```

```
let rec fold_right f l a =  
  match l with  
  [ ] -> a  
  | h::t -> f h (fold_right f t a)
```

Review: Fold

fold_left (+) 0 [1;2;3]

fold_left (+) 1 [2;3]

fold_left (+) 3 [3]

fold_left (+) 6 []

6

fold_right (+) [1;2;3] 0

1 + (fold_right (+) [2;3] 0)

1 + (2 + (fold_right (+) [3] 0))

1 + (2 + (3 (fold_right (+) [] 0)))

1 + (2 + (3 + 0)) 1 + (2 + 3)

1 + 5

6

OCaml Data

- So far, we've seen the following kinds of data
 - **Basic types** (int, float, char, string)
 - **Lists**
 - One kind of data structure
 - A list is either `[]` or `h::t`, deconstructed with pattern matching
 - **Tuples and Records**
 - Let you collect data together in fixed-size pieces
 - **Functions**
- How can we build other data structures?
 - Building everything from lists and tuples is awkward

(User-Defined) Variants

```
type gen =  
  |Int of int  
  |Str of string;;
```

```
let ls = [Int 10; Str "alice"]
```

```
let print_gen lst =  
  match lst with  
  |Int i->Printf.printf "%d\n" i  
  |Str s-> Printf.printf "%d\n" s
```

```
List.iter print_gen ls
```

Variants (full definition)

- Syntax

- **type** $t = C1$ [of $t1$] | ... | Cn [of tn]
- the Ci are called constructors

- Evaluation

- A constructor Ci is a value if it has no assoc. data
 - $Ci\ vi$ is a value if it does
- Destructing a value of type t is by pattern matching
 - patterns are constructors Ci with data components, if any

- Type Checking

- Ci [vi] : t [if vi has type ti]

Data Types: Variants with Data

```
type shape =  
  Rect of float * float  
  | Circle of float
```

```
let area s =  
  match s with  
    Rect (w, l) -> w *. l  
  | Circle r -> r *. r *. 3.14  
;;  
area (Rect (3.0, 4.0));; (* 12.0 *)  
area (Circle 3.0);;      (* 28.26 *)
```

```
[Rect (3.0, 4.0) ; Circle 3.0]. (* shape list*)
```

Quiz 1

```
type foo = ((string list) * int) list
```

Which one of the following could match type `foo`?

- A. `[("foo", "bar", 5)]`
- B. `[(["foo", "bar"], 6)]`
- C. `[([("foo", "bar")], 8)]`
- D. `[(["foo"; "bar"], 7)]`

Quiz 1

```
type foo = ((string list) * int) list
```

Which one of the following could match type `foo`?

- A. `[("foo", "bar", 5)] string * string * int) list`
- B. `[(["foo", "bar"], 6)] ((string*string) list*int) list`
- C. `[([("foo", "bar")], 8)] same as B`
- D. `[(["foo"; "bar"], 7)] (string list * int) list`

Quiz 2: What does this evaluate to?

```
type num = Int of int | Float of float;;  
let aux a =  
  match a with  
  | Int i -> i  
  | Float j -> int_of_float j  
;;  
aux (Float 5.0);;
```

- A. 5
- B. 2
- C. 5.0
- D. Type Error

Quiz 2: What does this evaluate to?

```
type num = Int of int | Float of float;;  
let aux a =  
  match a with  
  | Int i -> i  
  | Float j -> int_of_float j  
;;  
aux (Float 5.0);;
```

- A. 5
- B. 2
- C. 5.0
- D. Type Error

Option Type

```
type optional_int =  
  None  
  | Some of int
```

```
let divide x y =  
  if y != 0 then Some (x/y)  
  else None  
  
let string_of_opt o =  
  match o with  
    Some i -> string_of_int i  
  | None -> "nothing"
```

- Comparing to Java: **None** is like `null`, while **Some *i*** is like an **Integer (*i*)** object

Polymorphic Option Type

```
type 'a option =  
  Some of 'a  
| None
```

```
let opthd l =  
  match l with  
  [] -> None  
  | x::_ -> Some x
```

```
let p = opthd [];;      (* p = None *)  
let q = opthd [1;2];;  (* q = Some 1 *)  
let r = opthd ["a"];;  (* r = Some "a" *)
```

Quiz 3: What does this evaluate to?

```
let foo f = match f with
  None -> 42.0
  | Some n -> n +. 42.0
;;
foo 3.5;;
```

- A. 45.5
- B. 42.0
- C. Some 45.5
- D. Error

Quiz 3: What does this evaluate to?

```
let foo f = match f with
  None -> 42.0
  | Some n -> n +. 42.0
;;
foo 3.5;;    foo (Some 3.5)
```

- A. 45.5
- B. 42.0
- C. Some 45.5
- D. Error

Recursive Data Types: List

```
type 'a mylist =  
  Nil  
  | Cons of 'a * 'a mylist
```

```
let l = Cons (10, Cons (20, Cons (30, Nil)))
```

```
let rec len = function  
  Nil -> 0  
  | Cons (_, t) -> 1 + (len t)
```


Recursive Data Types: Binary Tree

```
type 'a tree =  
  Leaf  
  | Node 'a tree * 'a * 'a tree  
  
let empty = Leaf  
let t = Node(Leaf, 100, Node(Leaf, 200, Leaf))  
  
let rec sum t =  
  match t with  
  Leaf -> 0  
  | Node(l,v,r) -> sum l + v + sum r
```

OCaml Exceptions

```
exception My_exception of int
let f n =
  if n > 0 then
    raise (My_exception n)
  else
    raise (Failure "foo")
let bar n =
  try
    f n
  with My_exception n ->
    Printf.printf "Caught %d\n" n
  | Failure s ->
    Printf.printf "Caught %s\n" s
```

OCaml Exceptions: Useful Examples

- `failwith s`: Raises exception `Failure s` (`s` is a string).
- `Not_found`: Exception raised by library functions if the object does not exist
- `invalid_arg s`: Raises exception `Invalid_argument s`

```
let div x y =  
  if y = 0 then failwith "div by 0" else x/y;;  
  
let lst = [ (1, "alice") ; (2, "bob") ; (3, "cat") ] ;;  
let lookup key lst =  
  try  
    List.assoc key lst  
  with  
    Not_found -> "key does not exist"
```