

CMSC330: Higher-Order Functions in OCaml

Chris Kauffman

*Last Updated:
Mon Oct 2 11:22:36 PM EDT 2023*

Logistics

Assignments

- ▶ Project 3 Due Fri 06-Oct: Regex \rightarrow NFA \rightarrow DFA
- ▶ **Exam 1 on Thu 05-Oct**, covers topics through OCaml Pattern Matching

Reading: OCaml Docs <https://ocaml.org/docs>

- ▶ OCaml Docs: Lists
- ▶ OCaml Docs: Arrays

Demos `map` / `filter` / `iter` / `fold` on data structures

Goals: OCaml Data

- ▶ Pattern Matching and Linked Lists
- ▶ Higher-Order Functions in OCaml

Announcements

None

fun with Lambda Expressions

- ▶ Rather than `lambda`, OCaml provides anonymous functions via `fun` syntax
- ▶ Unlike in Python, `fun` has full syntax support for anything that appears in normal functions
- ▶ Note the equivalence below `let func a = ...` is short-hand for use of `let func = fun a -> ...`

```
1 let add1_stand x =                (* standard function syntax: add1_normal is *)
2   let xp1 = x+1 in                (* parameterized on x and remains unevaluated *)
3   xp1                             (* until x is given a concrete value *)
4 ;;
5
6 let add1_lambda =                 (* bind the name add1_lambda to ... *)
7   (fun x ->                        (* a function of 1 parameter named x. *)
8     let xp1 = x+1 in              (* Above standard syntax is "syntactic sugar" *)
9     xp1)                          (* for the "fun" version. *)
10 ;;
11
12 let eight = add1_stand 7;;        (* both versions of the function *)
13 let ate    = add1_lambda 7;;      (* behave identically *)
```

Common fun Use: Args to Higher-Order Functions

- ▶ Many higher-order functions require short, one-off function arguments for which fun can be useful

```
1 let evens list = (* even numbers *)
2   filter (fun n -> n mod 2 = 0) list
3 ;;
4 let shorter lim list = (* strings shortenr than lim *)
5   filter (fun s -> (String.length s) < lim) list
6 ;;
7 let betwixt min max list = (* elements between min/max *)
8   filter (fun e -> min < e && e < max) list
9 ;;
```

- ▶ If predicates are more than a couple lines, favor a named helper function with nicely formatted source code: **readability**

```
let is_some list = (* options that have some *)
  let pred opt = (* named predicate with *)
    match opt with (* formatted source code *)
    | Some a -> true (* that is boring but easy *)
    | None -> false (* on the eyes *)
  in
  filter pred list
;;
let is_some list = (* magnificent one-liner version... *)
  filter (fun opt -> match opt with Some a->true | None->false) list
;;
(* ...that will make you cry on later reading *)
```

First Class Functions Mean fun Everywhere

- ▶ fun most often associated with args to higher-order functions like filter BUT...
- ▶ A fun / lambda expression can be used anywhere a value is expected including but not limited to:
 - ▶ Top-level let bindings
 - ▶ Local let/in bindings
 - ▶ Elements of a arrays, lists, tuples
 - ▶ Values referred to by refs
 - ▶ Fields of records
- ▶ lambda_expr.ml demonstrates many of these
- ▶ Poke around in this file for a few minutes to see things like...

```
1 (* Demo function refs *)
2 let func_ref = ref (fun s -> s ^ " " ^ s);; (* a ref to a function *)
3 let bambam = !func_ref "bam";; (* call the ref'd function *)
4 func_ref := (fun s -> "!!!");; (* assign to new function *)
5 let exclaim = !func_ref "bam";; (* call the newly ref'd func *)
```

Families of Higher-Order Functions

- ▶ Recall the 4 major higher-order functions^a (shown below)
- ▶ OCaml provides LOTS of instances of these for its library of Data Structures (DS)

Pattern	Description	Library Functions
Filter	Select some elements from a DS (<code>'a -> bool</code>) -> <code>'a DS -> 'a DS</code>	<code>List.filter</code> , <code>Array.filter</code> <code>Map.filter</code> , <code>Hashtbl.filter</code>
Iterate	Perform side-effects on each element of a DS (<code>'a -> unit</code>) -> <code>'a DS -> unit</code>	<code>List.iter</code> , <code>Array.iter</code> <code>Queue.iter</code> , <code>Map.iter</code>
Map	Create a new DS with different elements, same size (<code>'a -> 'b</code>) -> <code>'a DS -> 'b DS</code>	<code>List.map</code> , <code>Array.map</code> <code>Map.map</code>
Fold/Reduce	Compute single value based on all DS elements (<code>'a -> 'b -> 'a</code>) -> <code>'a -> 'b DS -> 'a</code>	<code>List.fold_left</code> / <code>fold_right</code> <code>Array.fold_left</code> / <code>fold_right</code> <code>Queue.fold</code> , <code>Map.fold</code> <code>Hashtbl.fold</code>

^aIn some Object-Oriented programming circles, the [visitor pattern](#) affects the same idea as these higher-order functions: visit elements of a data structure and do something with them. FP makes this generally simpler and more flexible.

Exercise: iter visits all elements

- ▶ Frequently wish to visit each element of a data structure to do something for side-effects, e.g. printing
- ▶ Sometimes referred to as the *visitor pattern*
- ▶ `List.iter` is a higher-order function for iterating on lists

```
val List.iter : ('a -> unit) -> 'a list -> unit
```

- ▶ Sample uses: What happens in each case?

```
1 let ilist = [9; 5; 2; 6; 5; 1];;  
2 let silist = [("a",2); ("b",9); ("d",7)];;  
3 let ref_list = [ref 1.5; ref 3.6; ref 2.4; ref 7.1];;  
4  
5 (* Print all elems of an int list *)  
6 List.iter (fun i->printf "%d\n" i) ilist;;  
7  
8 (* Print all string,int pairs *)  
9 List.iter (fun (s,i)->printf "str: %s int: %d\n" s i) silist;;  
10  
11 (* Double the float referred to by each element *)  
12 List.iter (fun r-> r := !r *. 2.0) ref_list;;  
13  
14 (* Print all floats referred to *)  
15 List.iter (fun r-> printf "%f\n" !r) ref_list;;
```

- ▶ What would code for `iter` look like? Tail Recursive?

Answers: Iterate via iter

```
1 # let ilist = [9; 5; 2; 6; 5; 1];;
2 # List.iter (fun i->printf "%d\n" i) ilist;;
3 9
4 5
5 2
6 6
7 5
8 1
9 - : unit = ()
10
11 # let silist = [("a",2); ("b",9); ("d",7)];;
12 # List.iter (fun (s,i)->printf "str: %s int: %d\n" s i) silist;;
13 str: a int: 2
14 str: b int: 9
15 str: d int: 7
16 - : unit = ()
17
18 # let ref_list = [ref 1.5; ref 3.6; ref 2.4; ref 7.1];;
19 # List.iter (fun r-> r := !r *. 2.0) ref_list;;
20 - : unit = () (* refs are doubled *)
21
22 # List.iter (fun r-> printf "%f\n" !r) ref_list;;
23 - : unit = ()
24 3.000000
25 7.200000
26 4.800000
27 14.200000
```

```
(* Sample definition for iter:*)
(* tail recursive *)
let rec iter func list =
  match list with
  | [] -> ()
  | h::t -> func hd;
              iter func t
;;
```

map Creates a Transformed Data Structures

- ▶ Frequently want a new, different data structure, each element based on elements of an existing data structure
- ▶ *Transforms* 'a DS to a 'b DS with same size
 - ▶ **Not** mapping keys to values, different kind of map
- ▶ List.map is a higher-order function that transforms lists to other lists via an element transformation function

```
val List.map : ('a -> 'b) -> 'a list -> 'b list
```

- ▶ Example uses of List.map

```
1  # let ilist = [9; 5; 2; 6; 5; 1];;  
2  val ilist : int list = [9; 5; 2; 6; 5; 1]  
3  
4  # let doubled_list = List.map (fun n-> 2*n) ilist;;  
5  val doubled_list : int list = [18; 10; 4; 12; 10; 2]  
6  
7  # let as_strings_list = List.map string_of_int ilist;;  
8  val as_strings_list : string list = ["9"; "5"; "2"; "6"; "5"; "1"]
```

Exercise: Evaluate map Calls

- ▶ Code below makes use of `List.map` to transform a list to a different list
- ▶ Each uses a parameter function to transform single elements
- ▶ Determine the **value and type of the resulting list** in each case

```
1 let silist = [("a",2); ("b",9); ("d",7)];;
2 let ref_list = [ref 1.5; ref 3.6; ref 2.4; ref 7.1];;
3
4 (* Swap pair elements in result list *)
5 let swapped_list =
6   List.map (fun (s,i) -> (i,s)) silist;;
7
8 (* Extract only the first element of pairs in result list *)
9 let firstly_list =
10  List.map fst silist;;
11
12 (* Dereference all elements in the result list *)
13 let derefed_list =
14  List.map (!) ref_list;;
15
16 (* Form pairs of original value and its square *)
17 let with_square_list =
18  List.map (fun r-> (!r, !r *. !r)) ref_list;;
```

Answers: Evaluate map Calls

```
1 # let silist = [("a",2); ("b",9); ("d",7)];;
2 # let ref_list = [ref 1.5; ref 3.6; ref 2.4; ref 7.1];;
3
4 # let swapped_list = List.map (fun (s,i) -> (i,s)) silist;;
5 val swapped_list : (int * string) list =
6   [(2, "a"); (9, "b"); (7, "d")]
7
8 # let firstonly_list = List.map fst silist;;
9 val firstonly_list : string list =
10   ["a"; "b"; "d"]
11
12 # let derefed_list = List.map (!) ref_list;;
13 val derefed_list : float list =
14   [1.5; 3.6; 2.4; 7.1]
15
16 # let with_square_list = List.map (fun r-> (!r, !r *. !r)) ref_list;;
17 val with_square_list : (float * float) list =
18   [(1.5, 2.25); (3.6, 12.96); (2.4, 5.76); (7.1, 50.41)]
```

For completion, here is a simple definition for map:

```
19 (* Sample implementation of map: not tail recursive *)
20 let rec map trans list =
21   match list with
22   | []          -> []
23   | head::tail -> (trans head)::(map trans tail)
24 ;;
```

Compute a Value based on All Elements via fold

- ▶ Folding goes by several other names
 - ▶ **Reduce** all elements to a computed value OR
 - ▶ **Accumulate** all elements to a final result
- ▶ Folding is a very general operation: can write Iter, Filter, and Map via Folding and it is a **good exercise** to do so
- ▶ Will focus first on List.fold_left, then broaden

```
1 (*
2 val List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
3                               cur elem next  init  thelist  result
4 *)
5 (* sample implementation of fold_left *)
6 let fold_left func init list =
7   let rec help cur lst =
8     match lst with
9     | []          -> cur
10    | head::tail -> let next = func cur head in
11                     help next tail
12   in
13   help init list
14 ;;
```

Exercise: Uses of List.fold_left

Determine the values that get bound with each use of fold_left in the code below. *These are common use patterns for fold.*

```
1 let ilist = [9; 5; 2; 6; 5; 1];;  
2 let silist = [("a",2); ("b",9); ("d",7)];;  
3 let ref_list = [ref 1.5; ref 3.6; ref 2.4; ref 7.1];;  
4  
5 (* sum ints in the list *)  
6 let sum_oflist =  
7   List.fold_left (+) 0 ilist;;  
8  
9 (* sum squares in the list *)  
10 let sumsquares_oflist =  
11   List.fold_left (fun sum n-> sum + n*n) 0 ilist;;  
12  
13 (* concatenate all string in first elem of pairs *)  
14 let firststrings_oflist =  
15   List.fold_left (fun all (s,i)-> all^s) "" silist;;  
16  
17 (* product of all floats referred to in the list *)  
18 let product_oflist =  
19   List.fold_left (fun prod r-> prod *. !r) 1.0 ref_list;;  
20  
21 (* sum of truncating float refs to ints *)  
22 let truncsum_oflist =  
23   List.fold_left (fun sum r-> sum + (truncate !r)) 0 ref_list;;
```

Answers: Uses of List.fold_left

```
# let ilist = [9; 5; 2; 6; 5; 1];;  
# let silist = [("a",2); ("b",9); ("d",7)];;  
# let ref_list = [ref 1.5; ref 3.6; ref 2.4; ref 7.1];;  
  
# let sum_oflist = List.fold_left (+) 0 ilist;;  
val sum_oflist : int = 28  
  
# let sumsquares_oflist = List.fold_left (fun sum n-> sum + n*n) 0 ilist;;  
val sumsquares_oflist : int = 172  
  
# let firststrings_oflist = List.fold_left (fun all (s,i)-> all^s) "" silist;;  
val firststrings_oflist : string = "abd"  
  
# let product_oflist = List.fold_left (fun prod r-> prod *. !r) 1.0 ref_list;;  
val product_oflist : float = 92.016  
  
# let truncsum_oflist =  
    List.fold_left (fun sum r-> sum + (truncate !r)) 0 ref_list;;  
val truncsum_oflist : int = 13
```

Folded Values Can be Data Structures

- ▶ Folding can produce results of any kind including new lists
- ▶ Note that since the “motion” of `fold_left` left to right, the resulting lists below are in reverse order

```
1 # let ilist = [9; 5; 2; 6; 5; 1];;
2
3 (* Reverse a list via consing / fold *)
4 # let rev_ilist = List.fold_left (fun cur x-> x::cur) [] ilist ;;
5
6 val rev_ilist : int list = [1; 5; 6; 2; 5; 9]
7
8 (* Generate a list of all reversed sequential sub-lists *)
9 # let rev_seqlists =
10   List.fold_left (fun all x-> (x::(List.hd all))::all) [[]] ilist ;;
11 (*                               x::|list of prev|                               *)
12 (*                               |--longer list---|::all                           *)
13 val rev_seqlists : int list list =
14   [[1; 5; 6; 2; 5; 9];           (* all reversed *)
15    [5; 6; 2; 5; 9];             (* all but last reversed *)
16    [6; 2; 5; 9];                (* etc. *)
17    [2; 5; 9];                   (* 3rd::2nd::1st::init *)
18    [5; 9];                      (* 2nd::1st::init *)
19    [9];                         (* 1st::init *)
20    []                           (* init only *)
```


fold_left vs fold_right

Left-to-right folding, tail recursion, generates reverse ordered results

```
1 (* sample implementation of fold_left *)
2 let fold_left func init list =
3   let rec help cur lst =
4     match lst with
5     | []          -> cur
6     | head::tail ->
7       let next = func cur head in
8       help next tail
9   in
10  help init list
11 ;;
12
13 List.fold_left f init [e1; e2; ...; en]
14 = f (... (f (f init e1) e2) ...) en
15
16 # let nums = [1;2;3;4];;
17
18 # List.fold_left (+) 0 nums;;
19 - : int = 10
20
21 # List.fold_left (fun l e-> e::l) [] nums;;
22 - : int list = [4; 3; 2; 1]
```

Right-to-left folding, NOT tail recursive, allows in-order results

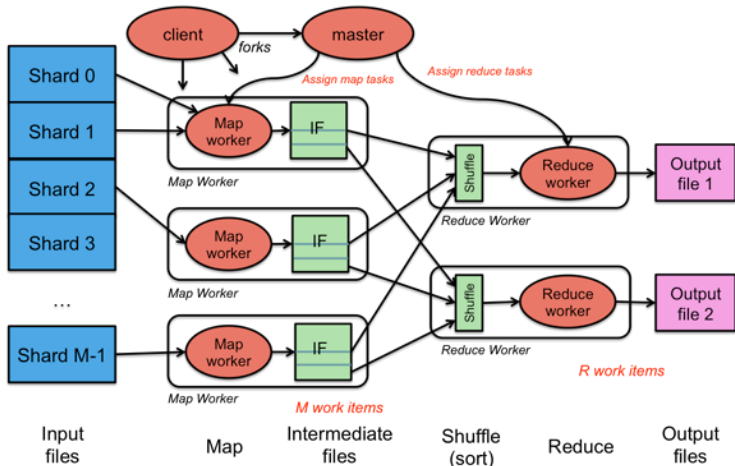
```
1 (* sample implementation of fold_right *)
2 let rec fold_right func list init =
3   match list with
4   | []          -> init
5   | head::tail ->
6     let rest = fold_right func tail init in
7     func head rest
8 ;;
9
10
11
12
13 List.fold_right f [e1; e2; ...; en] init
14 = f e1 (f e2 (... (f en init) ...))
15
16 # let nums = [1;2;3;4];;
17
18 # List.fold_right (+) nums 0;;
19 - : int = 10
20
21 # List.fold_right (fun e l-> e::l) nums [];;
22 - : int list = [1; 2; 3; 4]
```

(Optional): Distributed Map-Reduce

- ▶ Have seen that Map + Fold/Reduce are nice ideas to transform lists and computer answers
- ▶ In OCaml, tend to have *a list* of data that fits in memory, call these functions on that one list
- ▶ In the broader sense, a data list may instead be **extremely large**: a list of *millions of web pages* and their contents
- ▶ **Won't fit in the memory** or even on disk for a single computer
- ▶ A **Distributed Map-Reduce Framework** allows processing of large data collections on many connected computers
 - ▶ Apache Hadoop
 - ▶ Google MapReduce
- ▶ Specify a few functions that transform and reduce single data elements (*mapper* and *reducer* functions)
- ▶ Frameworks like Hadoop uses these functions to compute answers based on all data across multiple machines, all cooperating in the computation

Distributed Map-Reduce Schematic

- ▶ *Map*: function that computes category for a datum
- ▶ *Reduce*: function which computes a category's answer
- ▶ Individual Computers may be Map / Reduce / Both workers



Source: MapReduce A framework for large-scale parallel processing by Paul Krzyzanowski