CMSC330: Higher-Order Functions in OCaml

Chris Kauffman

Last Updated:
Tue Oct 3 09:27:48 AM 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

- 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 -> ...`

```
let add1_stand x =          (* standard function syntax: add1_normal is *)
  let xp1 = x+1 in          (* parameterized on x and remains unevaluated  *)
  xp1                       (* until x is given a concrete value *)
;;

let add1_lambda =           (* bind the name add1_lambda to ... *)
  (fun x ->                 (* a function of 1 parameter named x. *)
     let xp1 = x+1 in       (* Above standard syntax is "syntatic sugar" *)
     xp1)                   (* for the "fun" version. *)
;;

let eight = add1_stand 7;;  (* both versions of the function *)
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 `('a -> bool) -> 'a DS -> 'a DS`	`List.filter, Array.filter` `Map.filter, Hashtbl.filter`
Iterate	Perform side-effects on each element of a DS `('a -> unit) -> 'a DS -> unit`	`List.iter, Array.iter` `Queue.iter, Map.iter`
Map	Create a new DS with different elements, same size `('a -> 'b) -> 'a DS -> 'b DS`	`List.map, Array.map` `Map.map`

Fold/Reduce	Compute single value based on all DS elements
	`('a -> 'b -> 'a) -> 'a -> 'b DS -> 'a`
		`List.fold_left / fold_right`
		`Array.fold_left / fold_right`
		`Queue.fold, Map.fold`
		`Hashtbl.fold`

[a] In 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?

```
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];;

(* Print all elems of an int list *)
List.iter (fun i->printf "%d\n" i) ilist;;

(* Print all string,int pairs *)
List.iter (fun (s,i)->printf "str: %s  int: %d\n" s i) silist;;

(* Double the float referred to by each element *)
List.iter (fun r-> r := !r *. 2.0) ref_list;;

(* Print all floats referred to *)
List.iter (fun r-> printf "%f\n" !r) ref_list;;
```

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

Answers: Iterate via `iter`

```
# let ilist = [9; 5; 2; 6; 5; 1];;
# List.iter (fun i->printf "%d\n" i) ilist;;
9
5
2
6
5
1
- : unit = ()

# let silist = [("a",2); ("b",9); ("d",7)];;
# List.iter (fun (s,i)->printf "str: %s  int: %d\n" s i) silist;;
str: a  int: 2
str: b  int: 9
str: d  int: 7
- : unit = ()

# let ref_list = [ref 1.5; ref 3.6; ref 2.4;  ref 7.1];;
# List.iter (fun r-> r := !r *. 2.0) ref_list;;
- : unit = ()                                  (* refs are doubled *)

# List.iter (fun r-> printf "%f\n" !r) ref_list;;
- : unit = ()
3.000000
7.200000
4.800000
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

```
let silist = [("a",2); ("b",9); ("d",7)];;
let ref_list = [ref 1.5; ref 3.6; ref 2.4;  ref 7.1];;

(* Swap pair elements in result list *)
let swapped_list =
  List.map (fun (s,i) -> (i,s)) silist;;

(* Extract only the first element of pairs in result list *)
let firstonly_list =
  List.map fst silist;;

(* Dereference all elements in the result list *)
let derefed_list =
  List.map (!) ref_list;;

(* Form pairs of original value and its square *)
let with_square_list =
  List.map (fun r-> (!r, !r *. !r)) ref_list;;
```

Answers: Evaluate `map` Calls

```
# let silist = [("a",2); ("b",9); ("d",7)];;
# let ref_list = [ref 1.5; ref 3.6; ref 2.4;  ref 7.1];;

# let swapped_list = List.map (fun (s,i) -> (i,s)) silist;;
val swapped_list : (int * string) list =
  [(2, "a"); (9, "b"); (7, "d")]

# let firstonly_list = List.map fst silist;;
val firstonly_list : string list =
  ["a"; "b"; "d"]

# let derefed_list = List.map (!) ref_list;;
val derefed_list : float list =
  [1.5; 3.6; 2.4; 7.1]

# let with_square_list = List.map (fun r-> (!r, !r *. !r)) ref_list;;
val with_square_list : (float * float) list =
  [(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`:

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

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*.

```
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];;

(* sum ints in the list *)
let sum_oflist =
  List.fold_left (+) 0 ilist;;

(* sum squares in the list *)
let sumsquares_oflist =
  List.fold_left (fun sum n-> sum + n*n) 0 ilist;;

(* concatenate all string in first elem of pairs *)
let firststrings_oflist =
  List.fold_left (fun all (s,i)-> all^s) "" silist;;

(* product of all floats referred to in the list *)
let product_oflist =
  List.fold_left (fun prod r-> prod *. !r) 1.0 ref_list;;

(* sum of truncating float refs to ints *)
let truncsum_oflist =
  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

```
# let ilist = [9; 5; 2; 6; 5; 1];;

(* Reverse a list via consing / fold *)
# let rev_ilist = List.fold_left (fun cur x-> x::cur) [] ilist ;;

val rev_ilist : int list = [1; 5; 6; 2; 5; 9]

(* Generate a list of all reversed sequential sub-lists *)
# let rev_seqlists =
  List.fold_left (fun all x-> (x::(List.hd all))::all) [[]] ilist ;;
(*                             x::|list of prev|                    *)
(*                           |--longer list---|::all                 *)
val rev_seqlists : int list list =
  [[1; 5; 6; 2; 5; 9];          (* all reversed *)
   [5; 6; 2; 5; 9];             (* all but last reversed *)
   [6; 2; 5; 9];                (* etc. *)
   [2; 5; 9];                   (* 3rd::2nd::1st::init *)
   [5; 9];                      (* 2nd::1st::init *)
   [9];                         (* 1st::init *)
   []]                          (* init only *)
```

fold_left vs fold_right

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

```
(* sample implementation of fold_left *)
let fold_left func init list =
  let rec help cur lst =
    match lst with
    | []          -> cur
    | head::tail ->
       let next = func cur head in
       help next tail
  in
  help init list
;;

List.fold_left f init [e1; e2; ...; en]
 = f (... (f (f init e1) e2) ...) en

# let nums = [1;2;3;4];;

# List.fold_left (+) 0 nums;;
- : int = 10

# List.fold_left (fun l e-> e::l) [] nums;;
- : int list = [4; 3; 2; 1]
```

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

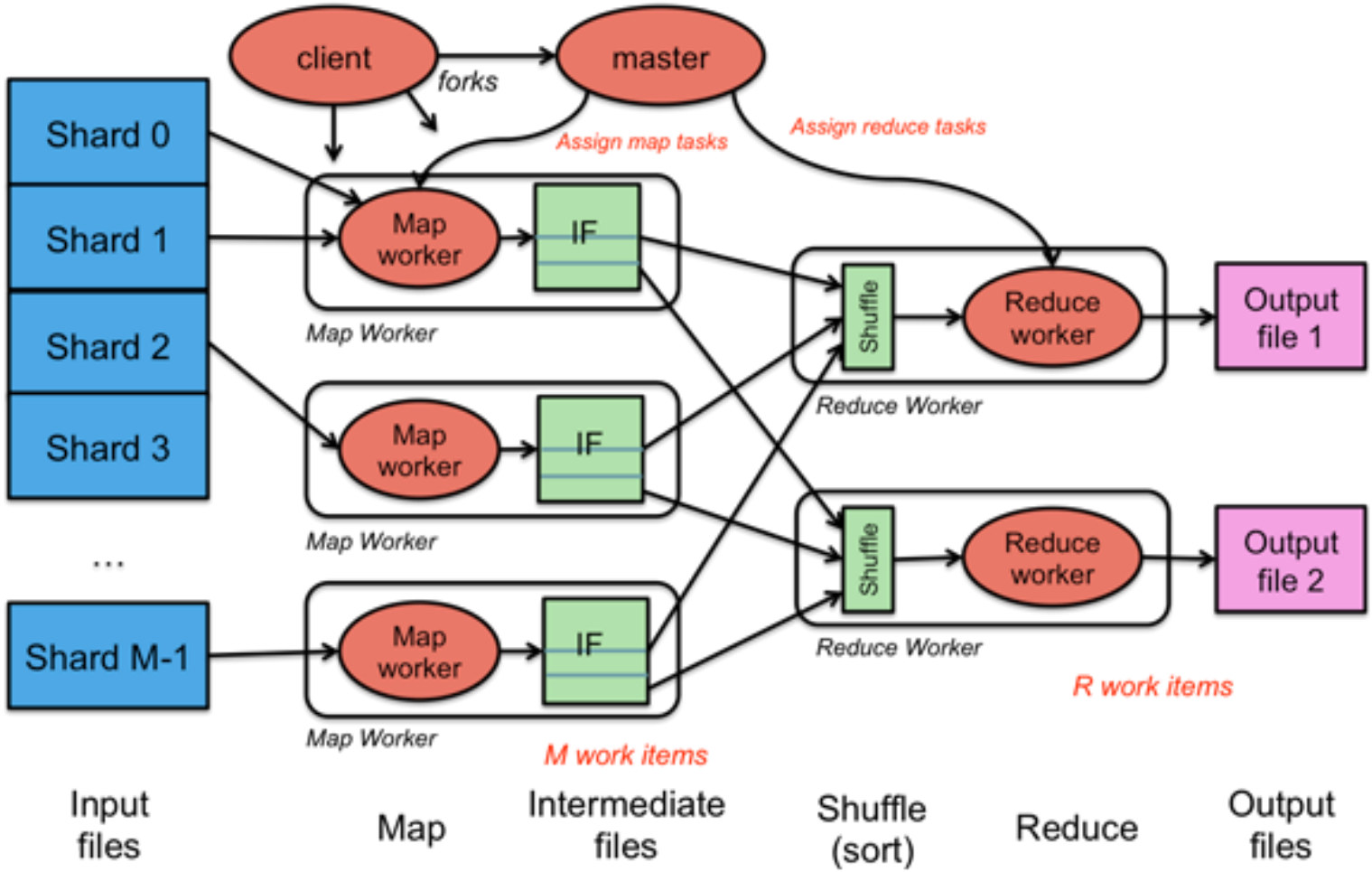
```
(* sample implementation of fold_right *)
let rec fold_right func list init =
  match list with
    | []          -> init
    | head::tail ->
       let rest = fold_right func tail init i
       func head rest
;;


List.fold_right f [e1; e2; ...; en] init
  = f e1 (f e2 (... (f en init) ...))

# let nums = [1;2;3;4];;

# List.fold_right (+) nums 0;;
- : int = 10

# List.fold_right (fun e l-> e::l) nums [];;
- : 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