# CSCI 2041: First Class Functions

Chris Kauffman

*Last Updated:*
*Thu Oct 18 22:42:48 CDT 2018*

# Logistics

## Reading

- OCaml System Manual: Ch 26: List and Array Modules, higher-order functions
- Practical OCaml: Ch 8

## Goals

- Functions as parameters
- Higher-order Functions
- Map / Reduce / Filter

## Assignment 3 `multimanager`

- Manage multiple lists
- Records to track lists/undo
- `option` to deal with editing
- Higher-order funcs for easy bulk operations
- Due Mon 10/22
- Test cases over the weekend

## Next Week

- Feedback Results
- Curried Functions
- Deep/Shallow Equality

# Exercise: Code Patterns on Lists

1. Describe the code structure that they share
2. Describe which parts differ between them
3. What is the shared purpose of the functions

```
let rec evens list =             (* all even ints in list *)
  match list with
  | []                        -> []
  | h::t when h mod 2 = 0 -> h::(evens t)
  | _::t                      -> evens t
;;

let rec shorter lim list =       (* all strings shortenr than lim *)
  match list with
  | []                                  -> []
  | h::t when String.length h < lim -> h::(shorter lim t)
  | _::t                                -> shorter lim t
;;

let rec betwixt min max list =   (* elements between min/max *)
  match list with
  | []                         -> []
  | h::t when min<h && h<max -> h::(betwixt min max t)
  | _::t                       -> betwixt min max t
```

# **Answers**: Code Patterns on Lists

1. Describe the code structure that they share
   - Each destructures the list and examines which elements satisfy some criteria.
   - List of the "true" elements results while "false" elements are excluded.
2. Describe which parts differ between them
   - The specific criteria for each function differs: evenness, string length, and within a range
   - The parameters associated with these conditions also change
3. What is the shared purpose of the functions
   - To **filter** a list down to elements for which some condition is true

Identifying a code pattern that is mostly copy-pasted creates an opportunity to write less and get more. OCaml provides a means to encapsulate this code pattern and others.

# Functions as Parameters

- OCaml features *1st class functions*
  - Functions can be passed as parameters to other functions
  - Functions can be returned as values from functions
  - Functions can be bound to names just as other values, global, local, or mutable names
- **Higher-order function**: function which takes other functions as parameters, i.e. a function OF functions
- Many code patterns can be encapsulated via higher-order functions

# Exercise: Basic Examples of Higher-Order Functions

Determine values bound to `a,b,c`

```
(* Higher-order function which
   applies func as a function to
   arg. *)
let apply func arg =
  func arg
;;

(* Simple arithmetic functions. *)
let incr n = n+1;;
let double n = 2*n;;

let a = apply incr 5;;
let b = apply double 5;;
let c = apply List.hd ["p";"q";"r"]
```

Determine values bound to `x,y,z`

```
(* Higher-order function taking two
   function paramters f1 and f2.
   Applies them in succession to
   arg. *)
let apply_both f1 f2 arg=
  let res1  = f1 arg in
  let res12 = f2 res1 in
  res12
;;

let x =
  apply_both incr double 10;;
let y =
  apply_both double incr 10;;
let z =
  apply_both List.tl List.hd ["p";
                              "q";
                              "r"];;
```

Determine the types for the two higher-order functions `apply` and `apply_both` shown below.

# **Answers:** Basic Examples of Higher-Order Functions

```
a = apply incr 5
  = (incr 5)
  = 6

b = apply double 5
  = (double 5)
  = 10

c = apply List.hd ["p";"q";"r"]
  = List.hd ["p";"q";"r"]
  = "p"

x = apply_both incr double 10
  = (double (incr 10))
  = (double 11)
  = 22

y = apply_both double incr 10
  = (incr (double 10))
  = (incr 20)
  = 21

z = apply_both List.tl List.hd ["p";"q";"r"]
  = (List.hd (List.tl ["p";"q";"r"]))
  = (List.hd ["q";"r"])
  = "q"
```

Function types:

```
let apply func arg = ...
val apply :
('a -> 'b) -> 'a -> 'b
|--func--|    arg   return

let apply_both f1 f2 arg = ...
val apply_both :
('a -> 'b) -> ('b -> 'c) -> 'a -> 'c
|---f1---|    |---f2---|    arg  return
```

Note that `apply_both` applies param func `f1` first then applies `f2` to that that result

# Exercise: Notation for Function Types

- Fill in the `???` entries in the table below dealing with types
- Entries deal with function param and return types
- Lower entries are higher-order functions
- Be able to describe in words what each entry means

| | Type Notation | #args | arg types | Return Type | Higher Order? |
|---|---|---|---|---|---|
| 1 | `int` | 0 | Not a function | `int` | No |
| 2 | `int -> string` | 1 | `???` | `string` | No |
| 3 | `int -> string -> int` | 2 | `???  + ???` | `???` | No |
| 4 | `???  -> bool` | 3 | `int + string + int` | `bool` | No |
| 5 | `(int -> string) -> int` | 1 | `(int -> string)` | `???` | Yes |
| 6 | `(int -> string) -> int -> bool` | ??? | `???` | `bool` | Yes |
| 7 | `???` | 2 | `int + (string-> int)` | `bool` | Yes |
| 8 | `(int -> string -> int) -> bool` | ??? | `???` | `bool` | Yes |

# **Answers**: Notation for Function Types

| | Type Notation | #args | arg types | Return Type | Higher Order? |
|---|---|---|---|---|---|
| 1 | `int` | 0 | Not a function | `int` | No |
| 2 | `int -> string` | 1 | `int` | `string` | No |
| 3 | `int -> string -> int` | 2 | `int + string` | `int` | No |
| 4 | `int -> string -> int -> bool` | 3 | `int + string + int` | `bool` | No |
| 5 | `(int -> string) -> int` | 1 | `(int -> string)` | `int` | Yes |
| 6 | `(int -> string) -> int -> bool` | 2 | `(int -> string) + int` | `bool` | Yes |
| 7 | `int -> (string -> int) -> bool` | 2 | `int + (string-> int)` | `bool` | Yes |
| 8 | `(int -> string -> int) -> bool` | 1 | `(int -> string-> int)` | `bool` | Yes |

## What about returning a function?

- Natural to wonder about type for returning a function. A good guess would be something like

  `int -> (string -> int)`

  for 1 `int` param and returning a `(string -> int)` function
- Will find that this instead written as

  `int -> string -> int`

  due to OCaml's **curried functions** (more later)

# Filtering as a Higher-order Function

- The following function captures the earlier code pattern

```
(* val filter : ('a -> bool) -> 'a list -> 'a list
   Higher-order function: pred is a function of a single element that
   returns a true/false value, often referred to as a "predicate".
   filter returns a all elements from list for which pred is true *)
let rec filter pred list =
  match list with
  | []                       -> []
  | h::t when (pred h)=true -> h::(filter pred t)
  | _::t                     -> filter pred t
```

- Allows expression of filtering functions using predicates

```
let evens list =                        (* even numbers *)
  let is_even n = n mod 2 = 0 in (* predicate: true for even ints *)
  filter is_even list                   (* call to filter with predicate *)
;;
let shorter lim list =                          (* strings w/ len < lim *)
  let short s = (String.length s) < lim in (* predicate *)
  filter short list                             (* call to filter *)
;;
let betwixt min max list =                  (* elements between min/max *)
  let betw e = min < e && e < max in    (* predicate *)
  filter betw list                          (* call to filter w/ predicate
;;
```

# Exercise: Use filter

- Define equivalent versions of the following functions
- Make use of filter in your solution

```
(* More functions that filter elements *)
let rec ordered list =      (* first pair elem < second *)
  match list with
  | []                   -> []
  | (a,b)::t when a < b -> (a,b)::(ordered t)
  | _::t                 -> ordered t
;;

let rec is_some list =      (* options that have some *)
  match list with
  | []          -> []
  | (Some a)::t -> (Some a)::(is_some t)
  | _::t        -> is_some t
;;
```

# **Answers**: Use filter

```
(* Definitions using filter higher-order function *)
let ordered list =                       (* first pair elem < second *)
  let pred (a,b) = a < b in
  filter pred list
;;

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
;;
```

# fun with Lambda Expressions

- OCaml's `fun` syntax allows one to "create" a function
- This function has no name and is referred to alternatively as
  - An anonymous function (e.g. no name)
  - A **lambda** expression (e.g. many Lisps use keyword `lambda` instead of `fun` to create functions)
  - Lambda (Greek letter $\lambda$) was used by Alonzo Church to represent "abstractions" (e.g. functions) in his calculus

```
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

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

- 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...

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

# Families of Higher-Order Functions

- Along with `filter`, there are several other common use patterns on data structures
- Most functional languages provide higher-order functions in their standard library for these use patterns on their built-in Data Structures (DS)
- Will discuss each of these: to harness the power of functional programming means **getting intimate with all of them**

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

# 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

```
# let ilist = [9; 5; 2; 6; 5; 1;];;
val ilist : int list = [9; 5; 2; 6; 5; 1]

# let doubled_list = List.map (fun n-> 2*n) ilist;;
val doubled_list : int list = [18; 10; 4; 12; 10; 2]

# let as_strings_list = List.map string_of_int ilist;;
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

```
(*
val List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
                      cur  elem  next   init   thelist   result
*)
(* 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
;;
```

# 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

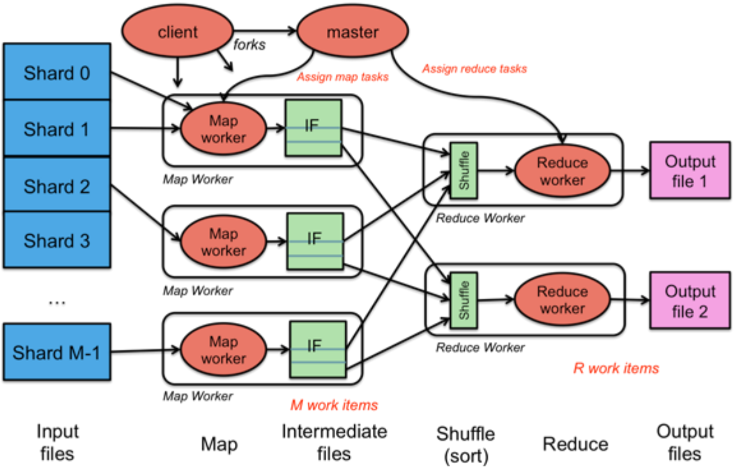
```
(* sample implementation of fold_right *)
let rec fold_right func list init =
  match list with
    | []          -> init
    | head::tail ->
       let rest = fold_right func tail init
       func head rest
;;




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]
```

# 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