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

```ocaml
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 shorter 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 $a, b, c$

```
1 (* Higher-order function which
2   applies func as a function to
3   arg. *)
4 let apply func arg =
5   func arg
6 ;;
7
8 (* Simple arithmetic functions. *)
9 let incr n = n+1;;
10 let double n = 2*n;;
11
12 let a = apply incr 5;;
13 let b = apply double 5;;
14 let c = apply List.hd ["p";"q";"r"]
```

Determine values bound to $x, y, z$

```
1 (* Higher-order function taking two
2   function parameters $f1$ and $f2$. 
3   Applies them in succession to
4   arg. *)
5 let apply_both $f1$ $f2$ arg=
6   let res1 = $f1$ arg in
7   let res12 = $f2$ res1 in
8   res12
9 ;;
10
11 let x =
12   apply_both incr double 10;;
13 let y =
14   apply_both double incr 10;;
15 let z =
16   apply_both List.tl List.hd ["p";
17     "q";
18     "r"];;
```

Determine the types for the two higher-order functions apply and apply_both shown below.
Answers: Basic Examples of Higher-Order Functions

```ocaml
code
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:

```ocaml
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 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

<table>
<thead>
<tr>
<th>Type Notation</th>
<th>#args</th>
<th>arg types</th>
<th>Return Type</th>
<th>Higher Order?</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 int</td>
<td>0</td>
<td>Not a function</td>
<td>int</td>
<td>No</td>
</tr>
<tr>
<td>2 int -&gt; string</td>
<td>1</td>
<td>???</td>
<td>string</td>
<td>No</td>
</tr>
<tr>
<td>3 int -&gt; string -&gt; int</td>
<td>2</td>
<td>??? + ???</td>
<td>???</td>
<td>No</td>
</tr>
<tr>
<td>4 ??? -&gt; bool</td>
<td>3</td>
<td>int + string + int</td>
<td>bool</td>
<td>No</td>
</tr>
<tr>
<td>5 (int -&gt; string) -&gt; int</td>
<td>1</td>
<td>(int -&gt; string)</td>
<td>???</td>
<td>Yes</td>
</tr>
<tr>
<td>6 (int -&gt; string) -&gt; int -&gt; bool</td>
<td>???</td>
<td>???</td>
<td>bool</td>
<td>Yes</td>
</tr>
<tr>
<td>7 ???</td>
<td>2</td>
<td>int + (string-&gt; int)</td>
<td>bool</td>
<td>Yes</td>
</tr>
<tr>
<td>8 (int -&gt; string -&gt; int) -&gt; bool</td>
<td>???</td>
<td>???</td>
<td>bool</td>
<td>Yes</td>
</tr>
</tbody>
</table>
## Answers: Notation for Function Types

<table>
<thead>
<tr>
<th>Type Notation</th>
<th>#args</th>
<th>arg types</th>
<th>Return Type</th>
<th>Higher Order?</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>0</td>
<td>Not a function</td>
<td>int</td>
<td>No</td>
</tr>
<tr>
<td>int -&gt; string</td>
<td>1</td>
<td>int</td>
<td>string</td>
<td>No</td>
</tr>
<tr>
<td>int -&gt; string -&gt; int</td>
<td>2</td>
<td>int + string</td>
<td>int</td>
<td>No</td>
</tr>
<tr>
<td>int -&gt; string -&gt; int -&gt; bool</td>
<td>3</td>
<td>int + string + int</td>
<td>bool</td>
<td>No</td>
</tr>
<tr>
<td>(int -&gt; string) -&gt; int</td>
<td>1</td>
<td>(int -&gt; string)</td>
<td>int</td>
<td>Yes</td>
</tr>
<tr>
<td>(int -&gt; string) -&gt; int -&gt; bool</td>
<td>2</td>
<td>(int -&gt; string) + int</td>
<td>bool</td>
<td>Yes</td>
</tr>
<tr>
<td>int -&gt; (string -&gt; int) -&gt; bool</td>
<td>2</td>
<td>int + (string-&gt; int)</td>
<td>bool</td>
<td>Yes</td>
</tr>
<tr>
<td>(int -&gt; string -&gt; int) -&gt; bool</td>
<td>1</td>
<td>(int -&gt; string-&gt; int)</td>
<td>bool</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### What about returning a function?

- Natural to wonder about type for returning a function. A good guess would be something like
  
  \[
  \text{int} \to (\text{string} \to \text{int})
  \]
  
  for 1 int param and returning a \((\text{string} \to \text{int})\) function

- Will find that this instead written as
  
  \[
  \text{int} \to \text{string} \to \text{int}
  \]
  
  due to OCaml’s **curried functions** (more later)
Filtering as a Higher-order Function

The following function captures the earlier code pattern

```ocaml
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

```ocaml
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

```ocaml
1 (* More functions that filter elements *)
2 let rec ordered list = (* first pair elem < second *)
3     match list with
4       | [] -> []
5       | (a,b)::t when a < b -> (a,b)::(ordered t)
6       | _::_:t -> ordered t
7    ;;

9 let rec is_some list = (* options that have some *)
10    match list with
11       | [] -> []
12       | (Some a)::t -> (Some a)::(is_some t)
13       | _::_:t -> is_some t
14    ;;
```
Answers: Use filter

1 (* Definitions using filter higher-order function *)
2 let ordered list = (* first pair elem < second *)
3     let pred (a,b) = a < b in
4     filter pred list
5 ;;

7 let is_some list = (* options that have some *)
8     let pred opt = (* named predicate with *)
9         match opt with (* formatted source code *)
10        | Some a -> true (* that is boring but easy *)
11        | None    -> false (* on the eyes *)
12        in
13     filter pred list
14 ;;
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 λ) was used by Alonzo Church to represent "abstractions" (e.g. functions) in his calculus

```ocaml
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 shorter 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 \texttt{fun} Everywhere

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

```ml
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

- 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**

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
<th>Library Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filter</td>
<td>Select some elements from a DS</td>
<td>List.filter, Array.filter</td>
</tr>
<tr>
<td></td>
<td>(<code>'a -&gt; bool</code>) -&gt; <code>'a DS</code> -&gt; <code>'a DS</code></td>
<td></td>
</tr>
<tr>
<td>Iterate</td>
<td>Perform side-effects on each element of a DS</td>
<td>List.iter, Array.iter</td>
</tr>
<tr>
<td></td>
<td>(<code>'a -&gt; unit</code>) -&gt; <code>'a DS</code> -&gt; unit</td>
<td>Queue.iter</td>
</tr>
<tr>
<td>Map</td>
<td>Create a new DS with different elements, same size</td>
<td>List.map, Array.map</td>
</tr>
<tr>
<td></td>
<td>(<code>'a -&gt; 'b</code>) -&gt; <code>'a DS</code> -&gt; <code>'b DS</code></td>
<td></td>
</tr>
<tr>
<td>Fold/Reduce</td>
<td>Compute single value based on all DS elements</td>
<td>List.fold_left / fold_right</td>
</tr>
<tr>
<td></td>
<td>(<code>'a -&gt; 'b</code>) -&gt; <code>'a DS</code> -&gt; <code>'a</code></td>
<td>Array.fold_left / fold_right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Queue.fold</td>
</tr>
</tbody>
</table>
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
  ```ocaml
code
val List.iter : ('a -> unit) -> 'a list -> unit
```
- Sample uses:
  What happens in each case?
  ```ocaml
code
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 like? Tail Recursive?
Answers: Iterate via iter

1 # let ilist = [9; 5; 2; 6; 5; 1];;;  (* Sample definition for iter:* )
2 # List.iter (fun i->printf "%d\n" i) ilist;;  (* tail recursive * )
3 9 let rec iter func list =
4 5  let rec iter func list =
5 2  match list with
6 6  | []  -> ()
7 5  | h::t  -> func hd;
8 1  iter func t
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
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
  ```plaintext
  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. (* Swap pair elements in result list *)
4. `let swapped_list = List.map (fun (s,i) -> (i,s)) silist;;`
5. (* Extract only the first element of pairs in result list *)
6. `let firstonly_list = List.map fst silist;;`
7. (* Dereference all elements in the result list *)
8. `let derefed_list = List.map (!) ref_list;;`
9. (* Form pairs of original value and its square *)
10. `let with_square_list = List.map (fun r-> (!r, !r *. !r)) ref_list;;`
Answers: Evaluate map Calls

```ocaml
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

```ocaml
(*
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.

```ocaml
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 =
    List.fold_left (+) 0 ilist;;
8
9  (* sum squares in the list *)
10 let sumsquares_oflist =
    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 =
    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 =
    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 =
    List.fold_left (fun sum r-> sum + (truncate !r)) 0 ref_list;;
```
Answers: Uses of List.fold_left

```ocaml
# 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 ;;

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 in
        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