Chapter 4: Higher-Order Programming

4.1. Higher-Order Functions

4.1.1 The Abstraction Principle

High-order functions: calling own function as an argument.
Refactor out functions and parameters functions on other functions

```
let double x = 2 * x
let square x = x * x
```

```
let quad x = double (double x)
let fourth x = square (square x)
```

Abstraction Principle: statements to avoid requiring something to be stated more than once. Instead, factor out the recurring pattern
Pipeline: the pipeline operator is a higher-order function

```
let double x = x * 2
let x = 6 |> double
```

```
utop # #use "example.ml";;
val double : int -> int = <fun>
val x : int = 12
```

example of pipeline function

```
let pipeline x f = f x
let (|>) = pipeline
let x = 5 |> double
```

textbook example. X : int = 10

Compose: we can write a function that composes two other functions

```
let double x = x*2
let square x = x * x
let compose a b c = a ( b c)
let square_then_double = compose double square
let answer1 = square_then_double 5
```

the compose function contains two other functions, it executes the parameter a

& b, then have the result executed as a's parameter. Similar to Calculus f(g(n)), passing n for g(n) first, then use the result to pass into f(n). For this equation, the square_then_double executes the square function, then the double function.

```
val square_then_double : int -> int = <fun>
val answer1 : int = 50
```

Both: write a function that applies two functions to the same argument and returns a pair as the result

```
let double x = x * 2
let square x = x * x
let both  f g x = (f x, g x)
let ds = both double square
let answer1 = ds 5
```

can see f=double function, g=square function, and x=passed in parameter to execute all the functions.

```
val ds : int -> int * int = <fun>
val answer1 : int * int = (10, 25)
```

the result will return (10, 25) as a pair of points

Cond: write a function that conditionally chooses which of the two functions to apply based on a predicate

```
let cond p f g x =
  if p x then f x else g x
```

switch to different functions depending on a predicate syntax/example

4.1.2 The Meaning of "Higher Order"

Higher order: used throughout logical and computer science
First order: quantification refers to primarily to the universal and existential quantifiers. Allows programmers to quantify over interested domains. Functions that operate on individual data elements
Second order: allows more power, quantify over properties of the domain. Assertions about individual elements

Third order: quantification over properties of properties
Fourth order: properties of properties of properties, and so fourth
High order: refers to all these logical that are more powerful than first order logic, logic can be expressed in second order logic.
Operate on functions

4.1.3 Famous Higher-order Functions

- map: transforms elements
- filter: eliminates elements
- fold: combines elements

4.2 Map

Map: allows programmers to individually transform each element of a list

```
(** [add1 lst] adds 1 to each element of [lst] *)
let rec add1 = function
  | [] -> []
  | h :: t -> (h + 1) :: add1 t

let lst1 = add1 [1; 2; 3]
```

```
val lst1 : int list = [2; 3; 4]
```

add1 function using simple recursions(version #1)

```
(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let rec concat_bang = function
  | [] -> []
  | h :: t -> (h ^ "!") :: concat_bang t

let lst2 = concat_bang ["sweet"; "salty"]
```

```
val lst2 : string list = ["sweet!"; "salty!"]
```

concatenate strings with “!” using simple recursions(version #1)

```ocaml
(** [add1 lst] adds 1 to each element of [lst] *)
let rec add1 = function
  | [] -> []
  | h :: t ->
    let f = fun x -> x + 1 in
    f h :: add1 t

(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let rec concat_bang = function
  | [] -> []
  | h :: t ->
    let f = fun x -> x ^ "!" in
    f h :: concat_bang t
```

(version #2): rewriting the two functions to make the difference more explicit

```ocaml
let rec add1' f = function
  | [] -> []
  | h :: t -> f h :: add1' f t

(** [add1 lst] adds 1 to each element of [lst] *)
let add1 = add1' (fun x -> x + 1)

let rec concat_bang' f = function
  | [] -> []
  | h :: t -> f h :: concat_bang' f t

(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let concat_bang = concat_bang' (fun x -> x ^ "!")
```

(version #3): abstracting the one helper function from the main function and creating an argument

```ocaml
let rec transform f = function
  | [] -> []
  | h :: t -> f h :: transform f t

(** [add1 lst] adds 1 to each element of [lst] *)
let add1 = transform (fun x -> x + 1)

(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let concat_bang = transform (fun x -> x ^ "!")
```

(version #4): transform method by apply a function to each element of the list

```
let rec map f = function
  | [] -> []
  | h :: t -> f h :: map f t

(** [add1 lst] adds 1 to each element of [lst] *)
let add1 = map (fun x -> x + 1)

(** [concat_bang lst] concatenates "!" to each element of [lst] *)
let concat_bang = map (fun x -> x ^ "!")
```

(version #5): final change, using the function word name of “map”

4.2.1 Side Effects

```
let rec map f = function
  | [] -> []
  | h :: t -> let h' = f h in h' :: map f t

let lst2 = map p [1; 2]
```

```
val map : ('a -> 'b) -> 'a list -> 'b list = <fu
```

```
1
```

```
2
```

```
val lst2 : int list = [2; 3]
```

used a let expression to cause the evaluation of the function application to occur before the recursive call

4.2.2 Map and Tail Recursion

```
let rec rev_map_aux f acc = function
  | [] -> acc
  | h :: t -> rev_map_aux f (f h :: acc) t

let rev_map f = rev_map_aux f []

let lst = rev_map (fun x -> x + 1) [1; 2; 3]
```

recursive map function that takes in user's input function, loop through the array to add 1

```
val lst : int list = [4; 3; 2]
```

output of the function above, standard library called this function List.rev_map, that returns the outputs in reverse order

```
let lst = List.rev (List.rev_map (fun x -> x + 1) [1; 2; 3])
```

```
val lst : int list = [2; 3; 4]
```

Use the List.rev library function to print the output in the "right" order, reverse output

4.2.3 Map in Other Languages

```
>>> print(list(map(lambda x: x + 1, [1, 2, 3])))
[2, 3, 4]
```

Python example

```
jshell> Stream.of(1, 2, 3).map(x -> x + 1).collect(Collectors.toList())
$1 ==> [2, 3, 4]
```

Java example

4.3 Filter

Filter: allows programmers to individually decide whether to keep or throw away each element of a list

Predicate: a function that returns a boolean
Cons: denoted by "::", constructs objects in memory, can be seen as " I will cons an element onto the list". OCaml adds an element onto the head of the list

```
(** [even n] is whether [n] is even. *)
let even n =
  n mod 2 = 0

(** [evens lst] is the sublist of [lst] containing only even numbers. *)
let rec evens = function
  | [] -> []
  | h :: t -> if even h then h :: evens t else evens t

let lst1 = evens [1; 2; 3; 4]
```

filter only the even numbers from a list

```
val lst1 : int list = [2; 4]
```

function returns only the even elements of the list

```
(** [odd n] is whether [n] is odd. *)
let odd n =
  n mod 2 <> 0

(** [odds lst] is the sublist of [lst] containing only odd numbers. *)
let rec odds = function
  | [] -> []
  | h :: t -> if odd h then h :: odds t else odds t

let lst2 = odds [1; 2; 3; 4]
```

filter only the odd numbers from a list

```
val lst2 : int list = [1; 3]
```

function returns only the odd elements of the list

```
let rec filter p = function
  | [] -> []
  | h :: t -> if p h then h :: filter p t else filter p t
```

change to map function, using p for predicate(a way of tests whether something is true or false)

```
let evens = filter even
let odds = filter odd
```

```
utop # evens [1; 2; 3; 4];;
- : int list = [2; 4]
```

filters even numbers using map function

4.3.1 Filter and Tail Recursion

```
let rec filter_aux p acc = function
  | [] -> acc
  | h :: t -> if p h then filter_aux p (h :: acc) t else filter_aux p acc t

let filter p = filter_aux p []

let lst = filter even [1; 2; 3; 4]
```

tail-recursive version of filter function

```
val lst : int list = [4; 2]
```

result of the above function

```
let rec filter_aux p acc = function
  | [] -> List.rev acc (* note the built-in reversal *)
  | h :: t -> if p h then filter_aux p (h :: acc) t else filter_aux p acc t

let filter p = filter_aux p []
```

add List.rev library function to “reverse” the output order

4.3.2 Filter in Other Languages

```
>>> print(list(filter(lambda x: x % 2 == 0, [1, 2, 3, 4])))
[2, 4]
```

Python

```
jshell> Stream.of(1, 2, 3, 4).filter(x -> x % 2 == 0).collect(Collectors.toList())
$1 ==> [2, 4]
```

Java

4.4 Fold

4.4.1 Combine

Combine: allows programmers to combine all the elements of a list

```
let rec combine init op = function
  | [] -> init
  | h :: t ->
    op h (combine init op t)
```

combining elements, using init and op, is the essential idea behind library functions known as fold

combining elements using init and op, is the essential idea of the library function fold

```
(** [sum lst] is the sum of all the elements of [lst]. *)
let rec sum = function
  | [] -> 0
  | h :: t -> h + sum t

let s = sum [1; 2; 3]
```

recursive function to sum all of the list example

```
val s : int = 6
```

```
(** [concat lst] is the concatenation of all the elements of [lst]. *)
let rec concat = function
  | [] -> ""
  | h :: t -> h ^ concat t

let c = concat ["a"; "b"; "c"]
```

recursive function to concat all of the string elements

```
val c : string = "abc"
```

```
let rec combine op init = function
  | [] -> init
  | h :: t -> op h (combine op init t)

let sum = combine ( + ) 0
let concat = combine ( ^ ) ""
```

combine function. [] value in the list gets replaced by init, each :: constructor gets replaced by op

4.4.2. Fold Right

Fold right: folds in values of the list from the right to the left, incorporating & combining them as it goes
Accumulator: denoted by “acc”, the result being accumulated as loop goes along

```
List.fold_right f [a;b;c] init
```

computes

```
f a (f b (f c init))
```

Accumulates an answer by

- repeatedly applying **f** to an element of list and “answer so far”
- folding in list elements “from the right”

List.fold_right, does the function from the right, accumulate it, then add it to the left value

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

fold_right example: f=function argument, lst=list, acc=accumulator. If the list is empty, then return the accumulator. Otherwise, apply through the fold_right f t acc function for all of t, then add it to the f h value. The reason the head executes last is because folding_right executes the right side, then add the left side, hence “folding_right = fold from the right”.   (F h = apply function f to value h)

****

```
List.fold_left f init [a;b;c]
```

computes

```
f (f (f init a) b) c
```

Accumulates an answer by

- repeatedly applying **f** to "answer so far" and an element of list
- folding in list elements “from the left”

List.fold_left, does the function from the left, accumulate it, then add it to the right value. Tail recursive

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

fold_left example: f acc h = accumulator including h, then recursively call the fold_left method on all list of t, add it to accumulator folding [1;2;3] with 0 and (+)

left to right: ((0+1)+2)+3 = 6
right to left: 1+(2+(3+0)) = 6

folding [1;2;3] with 0 and (-)

left to right: ((0-1)-2)-3 = -6
right to left: 1-(2-(3-0)) = 2

left vs. right: folding methods have different results depending on operator

4.4.3 Tail Recursion and Combine

```
let rec combine_tr f acc = function
  | [] -> acc
  | h :: t -> combine_tr f (f acc h) t  (* only real change *)
```

combine function with tail recursive. The function f is applied to the head element h and the accumulator acc before the recursive call is made, there won't be work remaining to be worked on after the call return

```
let sum = combine_tr ( + ) 0
Int
let s = sum [1; 2; 3]          val s : int = 6
```

example of calling the above function

4.4.4 Fold Left

```
let rec fold_left f acc = function
  | [] -> acc
  | h :: t -> fold_left f (f acc h) t

let sum = fold_left ( + ) 0
let concat = fold_left ( ^ ) ""
```

combine_tr function is also in the standard library under the name **List.fold_left**

4.4.5 Fold Left vs. Fold Right

Fold_left: proceeds from the left to the right, tail recursive
Fold_right: combines from the right to the left

```
List.fold_left;;
List.fold_right;;
```

4.4.7 Using Fold to implement other functions

```
let length lst =
  List.fold_left (fun acc _ -> acc + 1) 0 lst

let rev lst =
  List.fold_left (fun acc x -> x :: acc) [] lst

let map f lst =
  List.fold_right (fun x acc -> f x :: acc) lst []

let filter f lst =
  List.fold_right (fun x acc -> if f x then x :: acc else acc) lst []
```

4.4.8 Fold vs. Recursive vs. Library

Different ways to write function that manipulate lists:

1) directly use recursive functions that pattern match against empty list and agains cons
2) use fold functions
3) use other library functions

```
let rec lst_and_rec = function
  | [] -> true
  | h :: t -> h && lst_and_rec t

let lst_and_fold =
        List.fold_left (fun acc elt -> acc && elt) true

let lst_and_lib =
        List.for_all (fun x -> x)
```

4.5. Beyond Lists

4.5.1. Map on Trees

```
type 'a tree =
  | Leaf
  | Node of 'a * 'a tree * 'a tree
```

```
let t =
  Node (1,
       Node (2, Leaf, Leaf),
       Node (3, Leaf, Leaf))
```

defining a tree and a tree variable

```
let rec map f = function
  | Leaf -> Leaf
  | Node (v, l, r) -> Node (f v, map f l, map f r)

let add1 t = map succ t
```

recursive map function for a tree example. Example: add1 1;;

```
let rec fold acc f = function
  | Leaf -> acc
  | Node (v, l, r) -> f v (fold acc f l) (fold acc f r)
```

fold method for the recursion tree

```
let rec sum = function
  | Leaf -> 0
  | Node (v, l, r) -> v + sum l + sum r
```

sum method for tree. Example: sum t;;

4.5.2 Fold on Trees

```
let rec fold_tree f acc = function
  | Leaf -> acc
  | Node (v, l, r) -> f v (fold_tree f acc l) (fold_tree f acc r)
```

fold function on trees

```
let size t = fold_tree (fun _ l r -> 1 + l + r) 0 t
let depth t = fold_tree (fun _ l r -> 1 + max l r) 0 t
let preorder t = fold_tree (fun x l r -> [x] @ l @ r) [] t
```

using fold_tree method to implement tree functions

4.5.3. Filter on Trees

```
let rec filter_tree p = function
  | Leaf -> Leaf
  | Node (v, l, r) ->
    if p v then Node (v, filter_tree p l, filter_tree p r) else Leaf
```

```
val filter_tree : ('a -> bool) -> 'a tree -> 'a tree = <fun>
```

filter a node on trees, eliminates the children entirely.

4.6 Pipelining

```
let sum_sq n =
  let rec loop i sum =
    if i > n then sum
    else loop (i + 1) (sum + i * i)
  in loop 0 0
```

computing the sum of squares of the numbers from 0 - n example

```
let rec ( -- ) i j = if i > j then [] else i :: i + 1 -- j
let square x = x * x
let sum = List.fold_left ( + ) 0

let sum_sq n =
  0 -- n                (* [0;1;2;...;n]   *)
  |> List.map square    (* [0;1;4;...;n*n] *)
  |> sum                (*  0+1+4+...+n*n  *)
```

producing the same results using high-order functions and pipeline operator. First, function constructs a list containing all the elements of 0 … n, then uses the pipeline operator |> to pass that list through List.map square function, then the resulting list is pipelined through sum, which adds all the elements together.

4.7 Curring

```
let add x y = x + y
```

curried functions, takes two arguments of types t1 and t2. T1 -> t2 -> t3(int -> int -> int)

```
let add' t = fst t + snd t
```

use fst and snd tuple pattern, uncurried functions, takes two arguments of the tuples. T1 * t2 -> t3 (int * int -> int)

```
let add'' (x, y) = x + y
```

use tuple pattern, uncurried functions, takes tuple pattern in the function definition. x, y (int * int -> int)

```
let curry f x y = f (x, y)
let uncurry f (x, y) = f x y
 let uncurried_add = uncurry add
 let curried_add = curry add''
```

higher-order functions to do automatic conversions

Notes:

Function = match parameter with

Fold_right: acc is to the right of the list. (ex: fold_right f lst acc)

Fold_left: acc is to the left of the list. (Ex: fold_left acc lst).