Chapter 1

Higher Order Functions

> Higher Order Functions? More like lower suggestion inabilities
>
> Klef

1.1 Intro

We cover this topic in Ocaml so the examples here will be mostly written in Ocaml. A variation of this chapter written primarily in Python has been made as well.

1.2 Functions as we know them

Let us first define a function. A function is something that takes in input, or an argument, and then returns a value. As programmers, we typically think of functions as a thing that takes in multiple inputs and then returns a value. Technically, this is syntactic sugar for the most part (but that's a different chapter). The important idea now is that we have a process that has some sort of starting values, and then ends up with some other final value.

In the past, functions may have looked liked any of the following:

```
\\ java
int area(int length, int width){
    return length * width;
}
/* C */
int max(int* arr, int arr_length){
    int max = arr[0]
    for(int i =1; i < arr_length; i++)
        if arr[I] > max
            max = arr[i];
    return max;
}

# Ruby
def char-sum(str)
    sum = 0
    str..each_char{|i| sum += i.ord}
    sum
end
```

```
# python
def spam(x):
  return x - 1

(* OCaml *)
let circumference radius = 3.14 *. 2. *. radius

// Rust
fn foo(x:i32){
  let y = x + 2;
  y
}
```

In these functions, our inputs were things like data structures, or 'primitives'. Ultimately, our inputs were some sort of data type supported by the language. Our return value is the same, could be a data structure, could be a 'primitive', but ultimately some data type that is supported by the language.

This should hopefully all be straightforward, a review and pretty familiar. Notice there are 3 (I would say 4) parts of a function. We have the function name, the arguments, and the body (and then I would include the return type or value as well). Again this shouldn't be new, just wanted this here so we are all on the same page.

1.3 Functions as Data

Let's consider the C code:

```
1  int foo = 3;
2
3  int bar(){
4    return 3;
5  }
6
7  int main(){
8    int y = bar() + foo;
9    printf("%d\n",y);
10   return 1;
11 }
```

What exactly is happening here? We could say in line 1, we are allocating a segment in memory, binding that memory address to the human readable name `foo` and then storing 3 in that memory location.

What about lines 3-5? How is `bar` stored in memory? If we consider what is going on in the machine (Maybe recall from 216), then we know that any piece of data is just 1s and 0s stored at some memory address. The variable name helps us know which memory address we are storing things (so we don't have to remember what we stored at address 0x012f or something). When we want to then refer to that data, we use the memory address (variable name) and we retrieve that data. Why should a function be any different?

In this case, we are allocating a segment in memory (in the code part, rather than stack or heap), binding that memory segment to a human readable name `bar`, and then storing the code that represents the function to that location.

We can then treat the `bar` variable like we would treat any other variable. To be clear, they still follow the variable rules that other variables have. `3 + x` is only valid if `x` is an int or similar. So `bar` can only be used where we expect a function (`bar + 3` fails because we are treating `bar` as a number instead of a function).

1.4 Higher Programming

As we said, functions take in arguments that can be any data type supported by the language. A higher order programming language is one where functions themselves are considered a data type. We've seen this in OCaml, but let's take a deeper look at it now.

Let us consider the following C program:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int add1(int x){
  return x + 1;
}
int sub1(int x){
    return x - 1;
}

// return a function pointer
int* getfunc(){
  int (*funcs[2])(int) = {sub1, add1};
  return funcs[rand()%2];
}

// take in a function pointer
void apply(int f(int), int arg1){
  int ret = (*f)(arg1);
  printf("%d\n",ret);
}

int main(){
  int i;
  srand(time(NULL));
  for(i = 0; i < 5; i++){
    apply(getfunc(),3); //playing with pointers
  }
}
```

This program has one function that returns a function pointer and one function that takes in a function pointer. The idea of this is the basis of allowing functions to be treated as data. For most languages, we have the ability to bind variables to data.

```
int x = 3; // C, Java
y = 4 # Ruby
let z = 4.2;; (* OCaml *)
// idea
// variable = data
```

If we consider what is going on in the machine (maybe recall from 216), then we know that any piece of data is just 1s and 0s stored at some memory address. The variable name helps us know the memory address at which we are storing things (so we don't have to remember what we stored at address 0x012f or something). When we want to then refer to that data, we use the memory address (variable name) and we retrieve that data. Why should a function be any different? We previously saw a pointer to a function being passed around, which just means the pointer to a list of procedures that are associated with the function.

So in the case of higher order programming, we are allowing functions to be passed in as arguments or be returned as data.

Thus, we can say that a higher order function is one that takes in or returns another function. We can also avoid all these void pointers and casting and stuff in most functional languages like OCaml:

```
(* takes in a function)
let apply f x = f x;;
```

```
(* returns a function *)
let get_func = let add1 x = x + 1 in add1;;
```

1.5 Anonymous Functions

So we just said that we bind data to variables if we want to use them again. Sometimes though, we don't want to use them again, or we have no need to store a function for repeated use. So we have this idea of anonymous functions. It is anonymous because it has no variable name, which also means we cannot refer to it later. The syntax of an anonymous function is

```
(* add 1 *)
fun x -> x + 1
(* add *)
fun x y -> x + y
(* general syntax *)
(* fun var1:t1 var2:t2 ... varx:tx -> e:ty *)
(* has type (t1 -> t2 -> ... -> tx -> ty) *)
```

The difference between `2 + 3` and `let x = 2 + 3` corresponds to the difference between `fun x -> x + 1` and `let x = fun x -> x + 1`. This means that we can do the same thing by doing something like

```
2 + 3                    (* expression by itself, no variable *)
let x = 2 + 3            (* expression then bound to a variable *)
fun x -> x + 1           (* function by itself, no variable *)
let add1 = fun x -> x + 1 (* function bound to variable *)
```

This means `let add1 x = x + 1` is just syntactic sugar of `let add1 = fun x -> x + 1`. This is because OCaml and other functional programming languages are based on this thing called lambda calculus, which is another chapter. But if we think about our mathematical definition of a function, it is something that takes in 1 input and returns 1 output. So if each function should have 1 input, then what about something like `let plus x y = x + y`?

1.6 Partial Applications

Recall a section or something ago when we said that higher order functions can take in functions as arguments and return functions as return values. Consider:

```
let plus x y = x + y
(* int -> int -> int *)
```

We said earlier that functions have types, where the last thing in the type is the return value, and the first few items are the input types. We kinda lied. Let us consider:

```
let plus x y = x + y
(* int -> int -> int *)
let plus x = fun y -> x + y
(* int -> int -> int *)
(* int -> (int -> int) *)
```

This last function does have type `int -> int -> int` but consider what the syntax says. `plus` is a function that takes in an `int` but then returns a function that itself takes in an `int` and returns an `int`. Which means we can actually define plus as

```
let plus = fun x -> fun y -> x + y;;
```

If we can define functions like this then we can do things like

```
let plus = fun x -> fun y -> x + y
let add3 = plus 3 (* returning fun y -> 3 + y *)
add3 5 (* returns 8 *)
```

This is called a partial application of a function, or the process of currying. Not all functional languages support this unless the function is specifically defined as one which returns a function.

It is important to note here that you can only partially apply variables in the order used in the function declaration. That is a function like `let add = fun x -> fun y -> x + y` can only partially apply the x variable: `let add4 = add 4`. This is because we are technically doing something like `let add4 = fun y -> 4 + y`. If we wanted to partially apply the second parameter we would need to do something `let flip f x y = f y x in flip sub`.

To be more clear:

```
let sub x y = x - y
(* same as let sub = fun x -> fun y -> x - y *)
let minus3 = sub 3
(* let minus3 = fun y -> 3 - y *)
(* we cannot partially apply the second argument to sub unless we have a new function *)
(* we could make a sub specific function *)
let minus y = sub y 3
(* but let's make something generic *)
let flip f x y = f y x
(* let flip = fun f -> fun x -> fun y -> f y x *)
let sub3 = flip sub 3
(* let sub3 = fun y -> sub y 3 *)
```

So how does and currying supported language know what the values of variables are? Or how are partially applicated functions implemented? The answer lies with this idea of a closure, something thing that a Ruby Proc is.

1.7 Closures

If you look up the `Proc` object in the Ruby Docs, you will see that they call a Proc a closure. A closure is a way to create/bind something called a context or environment. Consider the following:

```
let and4 w x y z = w && x && y && z
(* and4 = fun w -> fun x -> fun y -> fun z -> w && x && y && z *)
let and3 = add4 true
(* and3 = fun x -> fun y -> fun z -> true && x && y && z *)
let and2 = and3 true
(* and2 = fun y -> fun z -> true && true && y && z *)
```

How does the language or machine know that you want to bind say variable w to `true`? To be honest, there is no magic, we just store the function, and then a list of key-value pairs of variables to values. This list of key-value pairs is called an environment. A closure is typically just a tuple of the function and the environment. Visually, a closure might look like the following:

```
let sub x y = x - y
let sub3 = sub 3
(* sub3 may look like
(function: fun y -> x - y, environment: [x:3])
*)
```

Very much like a `Proc` (because a `Proc` is a closure), a closure is not evaluated, or run until it is called. Thus, once made, the closure will not be modified. Thus the following would have no affect:

```
let sub x y = x - y
let x = 3
let sub3 = sub x
let x = 5
sub3 5 (* evaluates to -2 since 3 - 5 = -1 *)
```

Because the environment is not modified, and is evaluated with values that existed at the time of the closure's creation, we say that closures use static scope. This term is used in contrast with dynamic scope, where environment variables get updated to match typically top level variables. That is the above example would return `0` instead of `-2`.

1.8 Common HOFs

Part of the reason why higher order functions (HOFs) are so useful is because it allows us to be modular with out program design, and separate functions from other processes. To see this, consider the following that we say earlier:

```
let sub x y = x - y
let div x y = x / y
let mystery x y = (x*2)+(y*3)
let sub3 y = sub y 3
let div3 y = div y 3
let double y = mystery y 0
```

The functions `sub, div,` and `mystery` are all non-commutative (the order of inputs matter), so if we want to partially apply the second value, we need to write a new function that takes in a value to do so. Alternatively, we can just make a generic function that partially applies the second value so we don't need to ask for any input.

```
let flip f x y = f y x
let sub3 = flip sub 3
let div3 = flip div 3
let double = flip mystery 0
```

Being able to make similarly structured functions into a generic helps makes things modular, which is important to building good programs and designing good software. So the next sections are about common HOFs which will attempt to make a common function structure generic.

1.8.1 Map

Let us consider the following functions:

```
let rec double_items lst = match lst with
[] -> []
|h::t -> (h*2)::(double_items t)

let rec is_even lst = match lst with
[]->[]
|h::t -> (if h mod 2 = 0 then true else false)::(is_even t)

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

All of these functions aim to iterate through a list and modify each item. This is very common need and so instead of creating the above functions to do so, we may want to use this function called Map. Map will *map* the items from the input list (the domain) to a list of new item (co-domain). To take the above function and make it more generic, let us see that is the same across all of them:

```
let rec name lst = match lst with
[] -> []
|h::t -> (modify h)::(recursive_call t)
```

If we think about how we modify h, we will realize that we are just applying a function to h. Since it's the function that changes, we probably need to add it as a parameter. So adding this we should get

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

Fun fact: map actually exists in Ruby (`[1,2,3].map!{|x| x+1}`). Either way, in OCaml and other languages without imperative looping structures, this is a common recursive function that is needed and can be used to modify each item of a list by building a new list of the modified values (recall that everything in OCaml is immutable). Consider the code trace for adding 1 to each item in a `int list`.

```
let add1 x = x + 1 in map add1 [1;2;3]
(*
map add1 [1;2;3] = (add1 1)::(map add1 [2;3])
map add1 []2;3] = (add1 2)::(map add1 [3])
map add1 [3] = (add1 3)::(map add1 [])
map add1 [] = []
map add1 [1;2;3] = (add1 1)::(add1 2)::(add1 3)::[]
map add1 [1;2;3] = 2::3::4::[]
map add1 [1;2;3] = [2;3;4]
*)
```

1.8.2 Fold

While modifying each item in a list is useful, it is not the only common and useful list operation. Consider the following:

```
let rec concat lst = match lst with
[]-> ""
|h::t -> h^(concat t)

let rec sum lst = match lst with
[]-> 0
|h::t -> h+(sum t)

let rec product lst = match lst with
[]-> 1
|h::t -> h*(product t)

let rec ands lst = match lst with
[]-> true
|h::t -> h && (ands t)

let rec length lst = match lst with
[]-> 0
|_::t -> 1+(length t)
```

Here we want to take all the items in a list and return a single aggregate value. Doing this process is called folding and there are two common implementations. To start off, we will define define `fold_right`.

fold_right

So let us find out what each function has in common and then we can figure out what we need to add.

```
let rec name lst = match lst with
[] -> base_case
|h::t -> h operation (recursive_call t)
```

Taking a look at what we need, we need a base case, and we need an operation.

```
let rec fold_r f lst base = match lst with
[] -> base
|h::t -> f h (fold_r f t b)
```

Let us first talk about why it's `f h (fold_r f t b)`. In looking what was the same, we saw that it was `h operator (rec_call t)`. An operator is just a function so we are calling a function with 2 parameters: h and the recursive call `fold_r f t b`.

Some of you may also be wondering what about `lenth`? It doesn't use h, it uses a constant 1. My answer to this is to consider the type of the `f`. `f` is a function which takes in 2 parameters, h and the recursive call. I can easily just not use h in the body of `f`. Consider the following code trace:

```
1 let myfunc h rc = 1 + rc in
2 fold_r myfunc [1;2;3] 0
3 (*
4 fold_r myfunc [1;2;3] 0 = myfunc 1 (fold_r myfunc [2;3] 0)
5 fold_r myfunc [2;3] 0 = myfunc 2 (fold_r myfunc [3] 0)
6 fold_r myfunc [3] 0 = myfunc 3 (fold_r myfunc [] 0)
7 fold_r myfunc [] 0 = 0
8 fold_r myfunc [3] 0 = myfunc 3 0 = 1 + 0 = 1
9 fold_r myfunc [2;3] 0 = myfunc 2 1 = 1 + 1 = 2
10 fold_r myfunc [1;2;3] 0 = myfunc 1 3 = 1 + 2 = 3
11 *)
```

Notice here that lines 8-10 are just the stack frames all returning and propagating the return value up as stack frames are being popped off the stack. This also happens in `map` but there's something interesting with `fold` so I wanted to bring attention to it. That is, notice that if we send in a huge list we can potentially get a stackoverflow error or whatever OCaml's equivalent is. We can actually avoid this stack overflow and minimize the number of stack frames needed through the use of the other way to implement fold: `fold_left`. This is the default implementation of fold in most languages afaik so we typically just call this `fold`.

fold_left

See the next section (section 1.9) about why this we can minimize the number of stack frame, but since you already know how fold works, here is `fold_left`

```
let rec fold f a l = match l with
[] -> a
|h::t -> fold f (f a h) t
```

I used the variable a instead of base case or whatever because in this variation, the value is going to act as an accumulator. That is, this value is going to be constantly updated with each recursive call. Again see the next section for more about the accumulator. One thing to note is that this will evaluate the items in the list in the reverse order as `fold_right`. Consider the code trace for `fold_left`, then see them compared together.

```
(* take the sum of the list *)
let add x y = x + y in
fold add 0 [1;2;3;]
(*
fold add 0 [1;2;3] = fold add (add 0 1) [2;3] = fold add 1 [2;3]
fold add 1 [2;3]  = fold add (add 1 2) [3] = fold add 3 [3]
fold add 3 [3] = fold add (add 3 3) [] = fold add 6 []
fold add 6 [] = 6
*)
```

Now to compare the order of `fold_right` and `fold_left` we will use a non-commutative function: subtraction.

```
fold (-) 0 [1;2;3;]
(*
fold (-) 0 [1;2;3] = fold (-) ((-) 0 1) [2;3] = fold (-) -1 [2;3]
fold (-) -1 [2;3]  = fold (-) ((-) -1 2) [3] = fold (-) -3 [3]
fold (-) -3 [3] = fold add ((-) -3 3) [] = fold add -6 []
fold (-) -6 [] = -6
*)
(* compare this to fold_right *)
fold_r (-) [1;2;3] 0
```

```
(*
fold_r (-) [1;2;3] 0 = (-) 1 (fold_r (-) [2;3] 0)
fold_r (-) [2;3] 0 = (-) 2 (fold_r (-) [3] 0)
fold_r (-) [3] 0 = (-) 3 (fold_r (-) [] 0)
fold_r (-) [] 0 = 0
fold_r (-) [3] 0 = (-) 3 0 = 3
fold_r (-) [2;3] 0 = (-) 2 3 = -1
fold_r (-) [1;2;3] 0 = (-) 1 -1 = 2
*)
(* -6 != 2 *)
```

How interesting.

1.9 Tail Call Optimization

I was going to make this it's own chapter, but then had logistical questions so for now I decided against it and so I will just put this in the HOF chpater for some reason.

Let us take a trip back to our 216 days when we learned about stack frames and function calls. One thing I have noticed is that students get weird around recursion but I want you to consider the following

```
1  int fact1(int x){
2    if (x == 1)
3      return 1;
4    return -1
5  }
6  int fact2(int x){
7    if (x == 2)
8      return 2 * fact1(x-1);
9    return -1
10 }
11 int fact3(int x){
12   if (x == 3)
13     return 3 * fact2(x-1);
14   return -1
15 }
16 int fact4(int x){
17   if (x == 4)
18     return 4 * fact3(x-1);
19   return -1
20 }
```

Suppose we are on line 18. To evaluate what is returned, we have to call `fact3`, wait for it's return value, and then use that return value by multiplying it by 4. This is no different than it's recursive equivalent

```
int fact4(int x){}
    if (x == 1)
        return 1;
    if (x <= 4)
        return x * fact4(x-1);
    return -1;
}
```

The only difference is instead of calling a different function, waiting for it's return value, then using it's return value, we are instead calling ourself, waiting for a return value, then using that return value.

Great, so now that we know how recursion works, recall how a stack frame is created and pushed onto the memory stack when a function is called and then popped off the memory stack then the function returns. So the difference between something like the non-recursive `fact4` and the recursive `fact4`, is which function is being pushed to the stack.

So Consider what the stack looks like for the recursive `fact4` if we call `fact4(3)`

```
//Bottom of Stack//
3 // push argument on stack
---start of fact4(3) stack frame---
return 3 * fact4(2)
---end of fact4(3) stack frame---
2 // push argument on stack
---start of fact4(2) stack frame---
return 2 * fact4(1)
---end of fact4(2) stack frame---
2 // push argument on stack
---start of fact4(1) stack frame---
return 1
---end of fact4(1) stack frame---
```

Here we are pushing on stack frames when we call the recursive call. Then when finally get to out base case, we can then start popping values off. So popping off the texttttfact4(1) call would make the stack look like

```
//Bottom of Stack//
3 // push argument on stack
---start of fact4(3) stack frame---
return 3 * fact4(2)
---end of fact4(3) stack frame---
2 // push argument on stack
---start of fact4(2) stack frame---
return 2 * 1
---end of fact4(2) stack frame---
```

When you learned recursion, you probably learned about return values being propagated when teh function returns and this is how you can communicate values from one stack frame to another. This is definitely what happens, but notice that with something like recursive Fibonacci, you will get stack frames being added exponentially and you will get something like a `stackoverflow` error.

```
int fib(int x){
    if(x <= 1)
        return 1;
    return fib(x-1) + fib(x-2);
}
```

Here the number of stack frames increase at a rate of 2^x since each call to `fib` will push 2 more `fib` stack frames.

I think we can all agree that `Stackoverflow` errors are not good and if we can avoid them, we should. One way to avoid this is to use **tail call optimization** which would be something a compiler would use to optimize your code. To talk about tail call optimization, let us first talk about what the actual issue is.

The issue is that there are too many stack frames on the stack and then we run out of memory. There is 2 ways we can solve this issue: 1) add more memory or 2) pop things off the stack. The first solution doesn't really fix the issue, since memory is finite and we can just ask for something like `fib(10000000)`. The second solution has an issue because we need the old stack frames to exist. However, let us consider why we need the old stack frames.

In the previous example, we needed the old stack frame because before we could return, we needed the return value of a different stack frame.

```
//Bottom of Stack//
// calling fact4(3)
--------------------------------
return 3 * fact4(2)
//cannot return here since we need to first calculate fact4(2)
--------------------------------
return 2 * fact4(1)
```

```
//cannot return here since we need to first calculate fact4(1)
--------------------------------
return 1
--------------------------------
```

We said earlier that one way to pass in data from one stack frame to another is via the return value. However this is just communication from the callee to the caller. We can pass information from the caller to the callee by via argument values. So let consider this new factorial function:

```
int fact(int n, int a){
    if(n<=1)
        return a;
    return fact(n-1, n*a);
}
```

Notice that I added a new argument, a. This new parameter will allow the caller to send in data to the callee during the recursive call. Consider the following trace:

```
//Bottom of Stack//
// calling fact(3,1)
--------------------------------
// fact(3,1)
return fact(3-1,3*1) // fact(2,3)
--------------------------------
// fact(2,3)
return fact(2-1,2*3) // fact(1,6)
--------------------------------
// fact(1,6)
return 6
--------------------------------
```

Notice here that we get the same value, passing in the work of each stack frame into the next recursive call. What this means is that we no longer need to wait for the recursive call to finish, we can instead pop off stack frames once the recursive call happens.

```
//Bottom of Stack//
// calling fact(3,1)
--------------------------------
// fact(3,1)
return fact(3-1,3*1) // fact(2,3)

// we don't need the fact(3,1) stack frame so pop it off and push on fact(2,3) in it's place

//Bottom of Stack//
// calling fact(2,3)
--------------------------------
// fact(2,3)
return fact(2-1,2*3) // fact(1,6)

// we don't need the fact(2,3) stack frame so pop it off and push on fact(1,6) in it's place

//Bottom of Stack//
// calling fact(1,6)
--------------------------------
// fact(1,6)
return 6

// got the correct return value
```

So why is this called a tail call optimization and how to we make sure we are tail recursive? To answer this question let us look at the syntax of these recursive calls.

```
int nontailfact(int x)
    if (x == 1)
        return 1;
    return x * nontailfact(x-1);
}

int tailfact(int n, int a){
    if(n<=1)
        return a;
    return tailfact(n-1, n*a);
}
```

Where the one major difference is the number of arguments, tail optimization does not care about this. Remember that we care about the behavior of the recursive call. So if we notice the syntax around the recursive call, we can say that we care about what the last thing being calculated is during the recursive call. In the `nontailfact` the last thing being calculated is `x * nontailfact(x-1)`. In the `tailfact`, the last thing being calculated is `tailfact(n-1,n*a)`. This is purely a syntactical (visual) thing so we say that any statement that could be the last thing executed is in `tail position`. If the recursive call is in tail position, then we can take advantage of tail-call optimization.

Let us consider the tail position of some OCaml statements.

```
3
4
"a"
(* all of these statements are in tail position, since they are the last thing being evaluated *)

2 + 3
4 * 5
(* here 2,3,4,5 are not in tail position. The last thing calculated is 2*3, so we say the entire expression is in tail position. This is a tad confusing so let's see something clearer *).

[2+3;5*4;0-1]
(* here the last thing being evaluated is the creation of the list. So despite 2*3 being the last expression being evaluated, we still need to create the list so the entire expression is again in tail position *)

let x= 3 * 4 in x + 4
(* the last thing here is x+4 so the expression x + 4 is in tail position *)

let x = 3 + 4 in let x = 6 in 7
(* consider the syntax we used for a let binding: let v = e1:t1 in e2:t.
Here x = v, e1 = 3 + 4, and (let x = 6 in 7) is e2. Here at the top level, (or in broadest context), the expression in tail position is e2 or (let x = 6 in 7). If we changed our context to be more "zoomed in" or "jump in instead of jump over" then things in tail position would be just 7 *)
```

Again this is purely a syntactical thing which depends on the context of which parts of the expression will we consider. In an earlier section, we talked about `fold_right` and `fold_left`. They both do the same thing(ish), but one of them is tail recursive, and the other is not.