Chapter 1

Higher Order Functions

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

1.1 Intro

We cover this topic in Python so the examples here will be mostly written in Python. A variation of this chapter written primarily in OCaml will follow this one.

1.2 Functions as we know them

Let us first define a function. A function is something that takes in input, or a argument and then returns a value. As programmers, we typically think of functions as a thing that takes in multiple input and then returns a value. Technically this is syntactic sugar[1] for the most part but that's a different chapter. The important part is that we have this process that has some sort of starting values, and then ends up with some other final value.

in the past, functions may have looks 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;
}

# Python
def str_len(str)
  return len(str)
```

[1]syntactic sugar just means that the syntax looks pretty or sweet like sugar

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. The final note of this section is 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 Higher Order 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.

Let us consider the following C program:

```c
#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 # python
// 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 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? 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 just allowing functions to be passed in function data as arguments or be returned.

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

1.4 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. In python, we call these lambda functions. The syntax of a lambda function in python is

```
# add 1
lambda x: x + 1
(* add *)
lambda x,y: x + y
# general syntax
# lambda var1,var2,... varx: e
```

This is no different that just saying something like `2 + 3` instead of saying something like `x = 2 + 3`. This means that we can do the same thing by doing something like

```
2 + 3                 # expression by itself, no variable
x = 2 + 3             # expression then bound to a variable
lambda x: x + 1       # function by itself, no variable
add1 = lambda x: x + 1 # function bound to variable
```

Which means the following is just syntactic sugar.

```
def add1(x):
  return x + 1
# is syntactic sugar of
add1 = lambda x: x + 1
```

This is because the idea of functions as "first class data" is all 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 functions that have multiple inputs? (`max(x,y)`?)

1.5 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:

```
def plus(x,y):
  return x + y
```

We said earlier that functions have 1 input and 1 output. It seems here that this function has 2 inputs. We didn't lie, this is more syntactic sugar. Let us consider:

```
def plus(x,y):
  return x + y
def plus2(x):
  return lambda y: x + y
```

Here `plus2` 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

```
def plus():
  lambda x: lambda y: x + y
plus(2)(3)
```

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

```
plus = lambda x: lambda y: x + y
add3 = plus(3)
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 `add = lambda x: lambda y: x + y` can only partially apply the x variable: `add4 = add(4)`. This is because we are technically doing something like `add4 = lambda y: 4 + y`.

To be more clear:

```
def sub(x,y):
  return x - y
# same as let sub = lambda x : lambda y: x - y
def minus3(y):
  return sub(y,3)
# minus3 = fun y -> y - 3
```

So how does functional and currying supported languages know what the values of variables are? Or how are partially applied functions implemented? The answer lies with this idea of a closure.

1.6 Closures

A closure is a way to create/bind something called a context or environment. Consider the following:

```
def and4(w,x,y,z):
  return w and x and y and z
# and4 = lambda w: lambda x: lambda y: lambda z: w and x and y and z
def and3():
  return add4(True)
# and3 = lambda x: lambda y: lambda z: True x and y and z
def and2():
  return and3(True)
# and2 = lambda y: lambda z: True and True and y and 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:

```
def sub(x,y):
  return x - y
def sub3(x):
  return sub(x,3)
# sub3 may look like
# (function: lambda x: lambda y: x - y, environment: [y:3])
#
```

It is important to note how a language deals evaluates a closure. In some languages, the binding of variable to values occurs when the closure is created, and in others it occurs when the closure is called. In Python, the variable lookup is done when the closure is called. Consider the following Python code snippet

```python
x = 5
a = lambda y: x + y
print(a(5)) # prints 10
x = 3
print(a(5)) # prints 8
```

It is important to note, that when you have a local variable that is being bound, this will be bound independently of the global or higher scope variable. Consider

```python
sub = lambda x: lambda y: x - y
x = 3 # this is a different x than in the line above
subfrom3 = sub(x)
print(subfrom3(5)) #prints -2 (3-5 = -2)
x = 5
print(subfrom3(5)) # still -2
```

1.6.1 Nested Functions

It may be easier to visualize what a closure is when using nested functions.

```python
1 def outer_func(a):
2     def inner_func(b):
3         return a + b
4     return inner_func
```

From a previous chapter you may remember this example with a slight modifcation. In this case we are returning the function `inner_func`, rather than a call to the function (`inner_func(4)`. In this case, we could visualize the closure as lines 2-3 along with whatever value `'a'` is.

```python
c = outer_func(5) # c could be considered a variable that holds a closure
'''
If we want to visualize a closure, we could say that c is a variable that points to:
c = (def inner_func(b): return a + b, a = 5)
We can then call the closure say with
'''
d = c(5) # this will evaluate the closure substituting b with 5 and returning 10.
```

1.7 Common Higher Order Functions

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 saw earlier:

```python
def sub(x,y):
  return x - y
def div(x,y):
  return x / y
def mystery(x,y):
  return (x*2)+(y*3)
def sub3(y):
  return sub(y,3)
def div3(y):
  return div(y,3)
def double(y):
  return mystery(y,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. We will see this with two very common HOFs: map and fold/reduce.

1.7.1 Map

Let us consider the following functions:

```
def add1(lst):
  ret = []
  for x in lst:
    ret.append(x + 1)
  return ret

def times2(x):
  ret = []
  for x in lst:
    ret.append(x*2)
  return ret

def isEven(x):
  ret = []
  for x in lst:
    ret.append(x%2 == 0)
  return ret
```

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:

```
def common(lst):
  ret = []
  for x in lst:
    ret.append(___)
  return ret
```

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

```
def common(lst,f):
  ret = []
  for x in lst:
    ret.append(f(x))
  return ret
```

Fun fact: map actually exists in Python(`map(lambda x:  x + 1),[1,2,3])`. 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. Consider the code trace for adding 1 to each item in a `int list`.

```
1  def common(lst,f):
2    ret = []
3    for x in lst:
4      ret.append(f(x))
5    return ret
6
7  common([1,2,3],lambda x: x + 1)
8  '''
9  lst = [1,2,3]
10 f = lambda x: x + 1
11
12 starting at line 2 we set the initial value of ret
```

```
ret = []

then we iterate over the list of [1,2,3]
iteration 1:
  x = 1
  ret = []

  We then added an item to ret: ret.append(f(x))
  f(x) is the same as (lambda x: x + 1)(1)
  so ret.append(2) or since ret is [], [].append(2)
  so ret is now [2]

iteration 2:
  x = 2
  ret = [2]

  We then added an item to ret: ret.append(f(x))
  f(x) is the same as (lambda x: x + 1)(2)
  so ret.append(3) or since ret is [2], [2].append(3)
  so ret is now [2,3]

iteration 3:
  x = 3
  ret = [2,3]

  We then added an item to ret: ret.append(f(x))
  f(x) is the same as (lambda x: x + 1)(3)
  so ret.append(4) or since ret is [2,3], [2,3].append(4)
  so ret is now [2,3,4]

we now finish the loop and return ret: [2,3,4]
'''
```

1.7.2 Fold/Reduce

Depending on what language background you have, the following concept is called either fold or reduce.

Consider the following functions and what they have in common:

```
def sum(lst):
    final = 0
    for x in lst:
        final = final + x
    return final

def product(lst):
    final = 1
    for x in lst:
        final = final * x
    return final

def ands(lst):
    final = True
    for x in lst:
```

```
        final = final and x
    return final

def length(string):
    final = 0
    for x in string:
        final = final + 1
    return final

def concat(lst):
    final = ""
    for x in lst:
        final = final + x
    return final
```

These types of functions will go through a list and combine all the values in the list to a single value (it will reduce a bunch of things to one thing). Despite these looking a little more complex than the map examples, notice there is still much commonality between these examples.

Let us first consider what is exactly the same with all of these examples: a list (a string is just a list of characters), a final value, a looping construct that goes through the inputted list.

```
def common(lst):
    final = _
    for x in lst:
        final = ___# update final value
    return final
```

Now to make this generic structure more useful, we can first recognize that the initial value (where we first set `final` to a value) is dependent on the purpose of the function. In order to abstract this out, we can add it as a parameter:

```
def common(lst,initial):
    final = initial
    for x in lst:
        final = ___# update final value
    return final
```

Next we can consider that we need to update the final value. To do this, we need some sort of function that takes in two values: `final` and the value to combine with `final`. Notice that in every case except the penultimate one, this value to be added is an item in the list.

```
def common(lst,initial):
    final = initial
    for x in lst:
        final = update_func(final,x) # what to do when doing length?
    return final
```

Now `update_func` is not defined so we should add it as a parameter:

```
def common(lst,initial, update_func):
    final = initial
    for x in lst:
        final = update_func(final,x) # what to do when doing length?
    return final
```

Now if we wanted to write the above (sans `length` for now), we could do the following:

```
sum = common(lst,0,lambda x, y: x + y)
product = common(lst,1,lambda x, y: x * y)
ands = common(lst,True,lambda x, y: x and y)
concat = common(lst,"",lambda x, y: x + y)
```

Much more compact and notice that both `sum` and `concat` use the same function! This allows us to be more modular with our program design, as well as lead to less code duplication.

Now what about `length`? How do we use `common` to do this? There is no magic here, we can decide to just not use the parameter.

```
length = common(lst,0,lambda x, y: x + 1)
```

Now in python, this is called `reduce` and the order of the parameters are slightly different, but the concepts are the same:

```
sum = reduce(lambda x, y: x + y,lst,0)
product = reduce(lambda x, y: x * y,lst,1)
ands = reduce(lambda x, y: x and y,lst,True)
length = reduce(lambda x,y: x + 1,lst,0)
concat = reduce(lambda x, y: x + y,lst,"")
```

There are other common or useful HOF that exist (like `filter`), but this is a good introduction to HOFs and how to use them. As an exercise, try writing map in terms of `reduce`. For another exercise, look into `filter` and write that in terms of `reduce`.

1.8 Non List Map and Fold

Putting this here so it's written down. The above functions are typically done upon lists, but ultimately can be done upon any data structure. For example, map over a tree would go through each node in the tree and modify the value at each node. Similarly, a `reduce` over a tree wold visit every node in the tree and combine their values (eg. sum over a BST).