Chapter 1

Higher Order Functions

1.1 Intro

We cover this topic in Python so the examples here will be mostly written in Python.

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

\[
\text{\texttt{java}} \begin{aligned}
\text{int area(int length, int width)}
& \\{ \\
& \quad \text{return length * width;}
& \\}
\end{aligned}
\]

\[\begin{aligned}
\text{C} & \begin{aligned}
\text{int max(int* arr, int arr_length)}
& \\{ \\
& \quad \text{int max = arr[0]} \\
& \quad \text{for(int i =1; i < arr_length; i++)} \\
& \quad \quad \text{if arr[i] > max} \\
& \quad \quad \quad \text{max = arr[i];}
& \\}
\end{aligned}
\end{aligned}\]

\(^1\)syntactic sugar just means that the syntax looks pretty or sweet like sugar
return max;
}

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

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);
}
```
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.

```c
int x = 3; // C, Java
y = 4 # python
// idea
// variable = data
```

If we consider what is going on in the machine (Maybe recall from one osf), 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

```python
# 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

This is because actually 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? (\text{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:

```python
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:

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

Here plus2 is a function that takes in an \texttt{int} but then returns a function that itself takes in an \texttt{int} and returns an \texttt{int}. Which means we can actually define plus as

```python
def plus():
    lambda x: lambda y: x + y
```

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

```python
plus = lambda x: lambda y: x + y
add3 = plus3
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:
\[ x \times y \] can only partially apply the \( x \) variable: \( \text{add4} = \text{add}(4) \). This is because we are technically doing something like \( \text{add4} = \lambda y: 4 + y \).

To be more clear:

```python
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 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:

```python
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 \( \text{true} \)? 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:

```python
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 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:

```python
sub = lambda x: lambda y: x - y
x = 3
```
sub3 = sub(x)

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.

1.7 Map

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:

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 mod-
ular, which is important to building good programs and designing good software. We will
see this with a very common HOF: map.

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:

```python
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

```python
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 list.

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

common([1,2,3],lambda x: x + 1)

```

starting at line 2 we set the initial value of ret
```
ret = []
```
then we iterate over the list of [1,2,3]
CHAPTER 1. HIGHER ORDER FUNCTIONS

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]

There are other common or useful HOF that exist, but we will talk about them when we do OCaml.