Core Design Concepts Discussed:


# Functional Programming

## A Brief Introduction

**Harley Eades III**

There is a method of programming that is currently making a huge splash within the mainstream programming community called Functional Programming.  In this lecture, I introduce functional programming and discuss its benefits...

# Higher-Order Functions

Core Design Concepts:


```
let hoFunc1 (f : 'a -> 'b): 'c = ...
```

```
let hoFunc2 (x: 'a) : 'b -> 'c = ...
```

```
let hoFunc3 (x: 'a -> 'b) : 'c -> 'd = ...
```

Note:
- Make sure to use type annotations to make it clear that a function is being input or output.
- This also makes it easier to write higher-order functions.
- Remember: "Follow the types."

Functional programming is the use of functions to create reusable programming components called combinators. These combinators make heavy use of the concept of a higher-order function which are functions that take in a function as input or returns a function as output. Here we have three examples of the signatures of higher-order functions. The function hoFunc1 takes a function as an argument as its type indicates using the arrow type. On the opposite side, the function hoFunc2 returns a function as output. Lastly, the third function, hoFunc3, both takes in a function as input and produces a function as output.

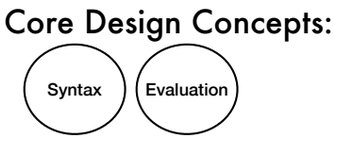
# Higher-Order Functions

Core Design Concepts:


```
let map (f : 'a -> 'b) (inputs : list 'a): list 'b =
  let rec map_helper (outputs : list b') (inputs' : list a'): list 'b
    match inputs' with
      []        -> outputs
      (x::rest) -> map_helper (f x :: outputs) rest
  in map_helper [] inputs
```

Here is an example combinator called map.  Notice that the types are completely generic in that we do not know the type of the elements of the list.  First, map takes in a function from a to b, then a list of inputs to this function, that is, a list of a's.  Then map applies the input function to each element of the list, and returns a new list of all the outputs of the input function.  Let's walk through it WALK THROUGH IT...

# Applying Higher-Order Functions



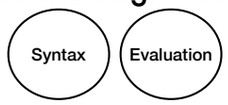

```
let addOne n = n + 1
```

```
map addOne [1,2,3] = [addOne 1, addOne 2, addOne 3]
```

Now lets see how we apply the map function.  First, we need a function we want to apply to each element of the list.  Here we define addOne that simply add's one to each element.  But, by using a very cool tool we can simplify this a bit...

# Applying Higher-Order Functions

Core Design Concepts:

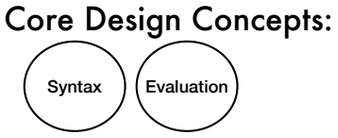


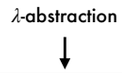

```
let addOne = fun n -> n + 1
```

```
map addOne [1,2,3] = [addOne 1, addOne 2, addOne 3]
```

We can redefine addOne using a....

# Applying Higher-Order Functions



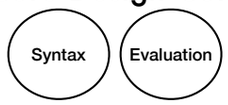

```
let addOne = fun n -> n + 1
```

```
map addOne [1,2,3] = [addOne 1, addOne 2, addOne 3]
```

lambda abstraction which is the same thing as an anonymous function.  When we apply map to addOne we get the same list, but why define addOne at all?...

# Applying Higher-Order Functions

Core Design Concepts:




```
map (fun n -> n + 1) [1,2,3] = [(fun n -> n + 1) 1,
                                (fun n -> n + 1) 2,
                                (fun n -> n + 1) 3]
                             = [1 + 1, 2 + 1, 3 + 1]
```

We simply place the lambda-abstraction where we had addOne before.  This is a very standard way of creating small functions to pass to combinators like map...

# Lambda's and Application

Core Design Concepts:


$\lambda$-abstraction:

```
fun x -> e
```

Application:

```
f a
```

In OCaml:

- Every function is a $\lambda$-abstraction.
- All functions are unary functions.

In OCaml every function is a lambda-abstraction and every function defined using a let or let-rec is also a lambda-abstraction.  This implies that all functions are unary functions, that is, they all take in a single argument, but you might be asking, how do you take in more than one input?...

# Lambda's and Application

Core Design Concepts:


$\lambda$-abstraction:

```
fun x -> e
```

Application:

```
f a
```

In OCaml:

- Every function is a $\lambda$-abstraction.
- All functions are unary functions.
- Thus, all functions are higher-order functions.

Functions that take in more than one input correspond to nested lambda-abstractions where the outer most abstraction takes in the first argument, but then returns a function that takes in the next argument and so forth.  A let simply allows us to give a function a name since lambda-abstractions are anonymous....

# Lambda's and Application

Core Design Concepts:


$\lambda$-abstraction:

```
fun x -> e
```

```
let myFun n m = e
```

```
let myFun = fun n -> (fun m -> e)
```

Application:

```
f a
```

```
let myFun n m k = e
```

```
let myFun = fun n ->
                (fun m ->
                   (fun k -> e))
```

Functions that take in more than one input correspond to nested lambda-abstractions where the outer most abstraction takes in the first argument, but then returns a function that takes in the next argument and so forth.  A let simply allows us to give a function a name since lambda-abstractions are anonymous....

# Lambda's and Application

Core Design Concepts:


$\lambda$-abstraction:

```
fun x -> e
```

```
let myFun (n: int) (m: int): int = e
```

What is the type of

```
myFun 1 : ?
```

Application:

```
f a
```

Every function being higher-order gives us a very simple, but powerful feature.  Consider myFun again, but now let me ask you, what is the type of myFun applied to 1?...

# Lambda's and Application

Core Design Concepts:


$\lambda$-abstraction:

```
fun x -> e
```

```
let myFun (n: int) (m: int): int = e
```

What is the type of

Application:

```
f a
```

```
myFun 1 : int -> int
```

it's int arrow int, but how did I arrive at this?  Well going back to what we've learned....

# Lambda's and Application

Core Design Concepts:


$\lambda$-abstraction:

```
fun x -> e
```

```
let myFun (n: int) (m: int): int = e
```

What is the type of

Application:

```
f a
```

```
myFun 1 : int -> int
```

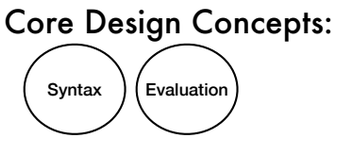
```
let myFun (n:int) (m: int): int = e
```

→

```
let myFun: int -> (int -> int) =
  fun (n: int) ->
    (fun (m: int) -> e)
```

we know that myFun is really a lambda-abstraction that returns another lambda-abstraction. So myFun's type is int arrow (int arrow int).  So if I apply myFun to only one argument, then the first int in it's type goes away, and so we are left with the return type of int arrow int.  So what can we do with partial application?...

# Applying Higher-Order Functions




```
let add n m = n + m
```

```
map (add 1) [1,2,3] = [(add 1) 1, (add 1) 2, (add 1) 3]
```

Consider again adding one to each element of our list, but this time we will us addition rather than the addOne function we used before.  Here we can partially apply add to 1 in the first argument to map, and obtain the exact same result. We will discuss higher-order functions and combinators more in class.