Lec 7: Higher-Order Functions

CS220: Programming Principles

Sang Kil Cha
Higher-Order Functions
What is a Higher-Order Function?

A function that manipulates functions: takes in a function as input, or returns a function as output.

This is naturally possible because functions are a value anyways!
Why Higher-Order Functions?

We can enhance our expressive power in programming!
Can You Find a Common Pattern?

```ml
let rec sumNum a b =
  if a > b then 0
  else a + sumNum (a + 1) b

let rec sumCubes a b =
  if a > b then 0
  else cube a + sumCubes (a + 1) b
```
Sigma Notation in Math

\[ \sum_{n=a}^{b} f(n) = f(a) + \cdots + f(b) \]

Regardless of the series being summed, we can formulate general results about sums with \( \sum \). Can we do the same with F#?
A function representing a sum of a series.

```fsharp
let rec sum term a next b =
    if a > b then 0
    else term a + sum term (next a) next b
```

Rewriting `sumNum` and `sumCube` with `sum`.

```fsharp
let inc n = n + 1
let sumNum a b = sum id a inc b
let sumCube a b = sum cube a inc b
```

---

_id_ is an identity function defined in F#. 

---

_id_ is an identity function defined in F#. 

---
Using Anonymous Functions

sumNum without inc.

```ml
let sumNum a b = sum id a (fun n -> n + 1) b
```
Defining Local Variables

Let us write a function:

\[ f(x, y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y) \]

We can simplify the function by letting \( a = 1 + xy \) and \( b = 1 - y \):

\[ f(x, y) = xa^2 + yb + ab \]
Defining Local Variables (cont’d)

```ml
let f x y =
  let a = 1 + x * y
  let b = 1 - y
  x * square a + y * b + a * b
```

With anonymous functions:

```ml
let f x y =
  (fun a b -> x * square a + y * b + a * b)
  (1 + x * y) (1 - y)
```
Example: Half-Interval Method

A root-finding method that repeatedly bisects an interval and then selects a subinterval in which a root must lie for further processing. This method is applicable for a continuous function $f$ defined on an interval $[a, b]$, where $f(a)$ and $f(b)$ have opposite signs.

---

Excerpt from Wikipedia.
Half-interval search.

```ml
let threshold = 0.001
let closeEnough x y = abs (x - y) < threshold
let avg x y = (x + y) / 2.0

let rec search f negPoint posPoint =
    let midPoint = avg negPoint posPoint
    if closeEnough negPoint posPoint then midPoint
    else
        let testValue = f midPoint
        if testValue > 0.0 then search f negPoint midPoint
        elif testValue < 0.0 then search f midPoint posPoint
        else midPoint
```

We cannot directly use this function because we don’t know the sign of \( f(x) \).
The final wrapper function for half-interval method.

```ml
let halfIntervalMethod f a b =
    let aValue = f a
    let bValue = f b
    if aValue > 0.0 && bValue < 0.0 then
        search f b a
    elif aValue < 0.0 && bValue > 0.0 then
        search f a b
    else
        failwith "Values are not of opposite sign"
```

halfIntervalMethod sin 2.0 4.0 // Returns 3.14 ...
failwith?

It is a special function that takes in a string and raises an exception, called Failure.

```
failwith

failwith: string -> 'a
```

Here, the type `'a` means that the resulting value type-checks with any type.
Example: Function Composition

Function composition is applying one function to the result of another. For example, 
\[(f \circ g)(x) = f(g(x)).\]

```
let compose f g = fun x -> f (g x)
let compose f g x = f (g x) // simpler

let squarePlusOne = compose inc square
squarePlusOne 10 // Returns 101
```
**Built-in Function Composition Operator**

Function composition operator (»).

```plaintext
let squarePlusOne = compose inc square
let squarePlusOne = square >> inc // order matters
```

How would you implement the operator (»)?

```plaintext
let (>>) f g x = g (f x)
```
Playing with Function Composition

Example: composition with partial application.

```ocaml
let add x y = x + y
let times x y = x * y
let add1Times5 = add 1 >> times 5
```

Example: applying a function twice.

```ocaml
let twice f = f >> f
let add10Twice = twice (add 10)
add10Twice 1 // Returns 21
```
Further Reading on Function Composition

See more examples from
Higher-Order Functions and Its Applications
Google’s MapReduce

MapReduce is a patented software framework introduced by Google to support distributed computing on large data sets on clusters of computers.

“Our abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages\(^1\).”

\(^1\)See the original paper appeared in OSDI 2004 by Dean et al.
val map: ('T -> 'U) -> 'T list -> 'U list
Map Example

```haskell
type Hero =
  | SuperMan
  | BatMan
  | SpiderMan

let heroes = [ SuperMan; BatMan; SpiderMan ] // Hero list

map isWearingMask heroes // [ false; true; true ]
map shirtColor heroes // [ Blue; Black; Red ]
```

First, try to write functions without high-order functions (without using `map`). For example, write a function `checkWearingMask` that takes in a list of heroes, and returns a list of booleans.
Implementing Map

let rec map f = function
  | [] -> []
  | hd :: tl -> (f hd) :: (map f tl)

Is it tail-recursive?
Fold

val fold: ('State -> 'T -> 'State) -> 'State -> 'T list -> 'State

• The 'State is often called **accumulator** (or acc).
• Evaluate elements from left to right.
Fold Example

```
// List of heroes
let heroes = [ SuperMan; BatMan; SpiderMan ]
fold sumOfCapes 0 heroes // 2
```
Implementing Fold

let rec fold f acc = function
| [] -> acc
| hd :: tl -> fold f (f acc hd) tl
FoldBack: Folding from Right

val foldBack:
('T -> 'State -> 'State) -> 'T list -> 'State -> 'State

• Evaluate elements from right to left.
• Often called foldr (fold right).
FoldBack Example

// List of heroes
let heroes = [ SuperMan; BatMan; SpiderMan ]

fold sumOfCapes 0 heroes // 2
foldBack sumOfCapes heroes 0 // 2

Does the order matter?
Folding Order

```javascript
let lst = [1; 2; 3; 4; 5]
fold (+) 0 lst // 15
foldBack (+) lst 0 // 15
fold (-) 15 lst // 0
foldBack (-) lst 15 // -12 Why?
```
Performance: Fold vs. FoldBack

Which one is more efficient? and Why?
Reduce

val reduce: ('T -> 'T -> 'T) -> 'T list -> 'T

• Somewhat similar to fold.
• Evaluate pairs of elements from left to right: $f(\ldots f(f_i_0 i_1) i_2\ldots) i_N$.
• For example, reduce (+) [1; 2; 3] // Should return 6
Map and Reduce

Two steps:

- **Map**: Apply a function $f$ in *parallel* to a list of some data.
- **Reduce**: Reduce all the results with a function $g$ to obtain a final result.

```
data // a list or sequence
|> map f
|> reduce g```

In-Class Activity #8

Implement your own `reduce` function by modifying the `myfunc` function. We will use the `MyList` type implemented in the previous activity.
More Examples

- map: (’T -> ’U) -> ’T list -> ’U list
- fold: (’State -> ’T -> ’State) -> ’State -> ’T list -> ’State
- foldBack: (’T -> ’State -> ’State) -> ’T list -> ’State -> ’State
- reduce: (’T -> ’T -> ’T) -> ’T list -> ’T
- filter: (’T -> bool) -> ’T list -> ’T list
- forall: (’T -> bool) -> ’T list -> bool
- collect: (’T -> ’U list) -> ’T list -> ’U list
- exists: (’T -> bool) -> ’T list -> bool
- And many more.

These functions take in an action (a function) as input.
Built-in Support!

All the above functions are defined as `List.xxx`, where `xxx` is the name of a higher-order function. See https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/fsharp-collection-types for the complete list.
Exception
Defining Exceptions

```ml
exception MyException
let f x =
  if x > 0 then x - 1
  else raise MyException

exception AnotherException of string
let g x =
  if x > 0 then x - 1
  else raise (AnotherException "message")
```
How about **failwith**?

`failwith` is a function that raises a predefined exception (`System.Exception`). There are several other error handling functions in F#:

- `failwith`
- `invalidArg`
- `nullArg`
- `invalidOp`

Handling Exceptions

Use a try .. with statement.

```ml
let x =
  try f (-1)
  with MyException -> // do something here.

let y =
  try g (-1)
  with AnotherException s -> // do something here with s.
```
When a function raises multiple exceptions.

```ml
let z =
    try someFunction 10
    with
        | MyException -> // case 1.
        | AnotherException s -> // case 2.
```
Exception vs. Option

It is preferred to use the Option (or Error) type over Exception. Why? because exceptions are slow in F#. Use exception only when you are dealing with a fatal case: cases where you don’t need to recover from the error.
Conclusion
1. Higher-order functions expand our expressive power.
2. Using higher-order functions is so common in functional languages, and F# provides built-in higher-order functions for manipulating collections.
Question?