Lec 9: Higher-Order Functions

CS220: Programming Principles

Sang Kil Cha

Higher-Order Functions

Motivation

We want to make our code concise, thereby “easy to read”.

Motivation

We want to make our code concise, thereby “easy to read”.

- We already learned how to abstract our ideas by writing a function.

Motivation

We want to make our code concise, thereby “easy to read”.

- We already learned how to abstract our ideas by writing a function.
- But what if we want to express complex algorithms?

Motivation

We want to make our code concise, thereby “easy to read”.

- We already learned how to abstract our ideas by writing a function.
- But what if we want to express complex algorithms?

Writing a single function for a complex algorithm is not desirable because it is ***not*** easy to read. Instead, we should split our ideas into smaller pieces (i.e., functions) and combine them.

Decomposition

Decomposition (a.k.a. factoring) is breaking a complex problem or system into parts that are easier to conceive, understand, program, and maintain[1].

[1] Wikipedia: https://en.wikipedia.org/wiki/Decomposition_(computer_science)

Decomposition

Decomposition (a.k.a. factoring) is breaking a complex problem or system into parts that are easier to conceive, understand, program, and maintain[1].

Higher-order functions help in factoring your code.

[1] Wikipedia: `https://en.wikipedia.org/wiki/Decomposition_(computer_science)`

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!

Attendance Check

Note:

1. This slide appears at random time during the class.
2. This link is only valid for a few minutes.
3. We don’t accept late responses.

Can You Find a Common Pattern?

```
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 + subCubes (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.

```
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`.

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

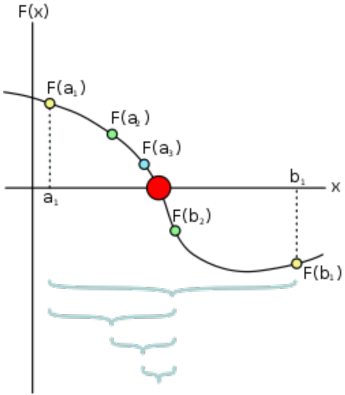
`id` is an identity function defined in F#.

Using Anonymous Functions

sumNum without `inc`.

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

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[a]. This method is applicable for a continuous function f defined on an interval $[a, b]$, where $f(a)$ and $f(b)$ have opposite signs.

[a] Excerpt from Wikipedia.

Half-interval search.

```
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.

```
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 ...
```

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 (»).

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

How would you implement the operator (»)?

```
let (>>) f g x = g (f x)
```

Playing with Function Composition

Example: composition with partial application.

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

Example: applying a function twice.

```
let twice f = f >> f
let addTenTwice = twice (add 10)
addTenTwice 1 // Returns 21
```

Further Reading on Function Composition

See more examples from `https://fsharpforfunandprofit.com/posts/function-composition/`.

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.

Function and Let-bindings

Function is a powerful abstraction mechanism, and it can even replace let-bindings!

Example

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 $b = 1 - y$:

$$f(x, y) = xa^2 + yb + ab$$

Example (cont'd)

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

With anonymous functions:

```
let f x y =
  (fun a b -> x * square a + y * b + a * b)
    (1 + x * y) (1 - y)
```

Functions as First-Class Citizens

- Functions are the most crucial component of functional programming.
- Functions are values.
- Functions can be passed as arguments to other functions.
- Functions can be returned as results from other functions.

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[2].”

[2] See the original paper appeared in OSDI 2004 by Dean et al.

Map

```
val map:  ('T -> 'U) -> 'T list -> 'U list
```

Map Example

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

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

map isWearingMask heroes // [ false; true; true ]
map (fun h -> isWearingMask h) heroes // bad style
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?

In-Class Activity #09

Preparation

We are going to use the same git repository as before. Just in case you don't have it, clone the repository using the following command.

1. Clone the repository to your machine.

   ```
   - git clone https://github.com/KAIST-CS220/CS220-Main.git
   ```

2. Move in to the directory `CS220-Main/Activities`

   ```
   - cd CS220-Main
   - cd Activities
   ```

Tail-Recursive `map`

- Implement `isWearingMask` and `shirtColor` functions.
- Write your own `map` function that is tail-recursive.

Exception

Defining Exceptions

```
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`

See `https://fsharpforfunandprofit.com/posts/exceptions/` for more information.

Handling Exceptions

Use a `try` .. `with` statement.

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

let y =
  try g (-1)
  with AnotherException s -> // do something here with s.
```

Handling Exceptions (cont’d)

When a function raises multiple exceptions.

```
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. Functions are first-class citizens in F# and many other functional languages.

Question?