Lec 10: Higher-Order Functions (2)

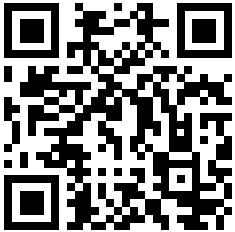
CS220: Programming Principles

Sang Kil Cha

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.



Higher-Order Functions

In functional programming languages, functions are first-class values. This means that functions can be passed as arguments to other functions, returned as values from other functions, etc.

Higher-Order Functions

In functional programming languages, functions are first-class values. This means that functions can be passed as arguments to other functions, returned as values from other functions, etc.

There are ***common patterns*** of using higher-order functions. We will learn some of them in this lecture.

Recap: Map

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

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

```
let lst = [ 1; 2; 3; 4; 5 ]

fold (+) 0 lst // 15
foldBack (+) lst 0 // 15

fold (-) 15 lst // 0
foldBack (-) lst 15 // -12  Why?
```

Reduce

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

- Somewhat similar to `fold`.
- Evaluate pairs of elements from left to right: $f(...\ f\ (f\ i_0\ i_1)\ i_2 ...)\ i_N$.
- 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 #10

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`

Problem 1: FoldBack

Implement both `sumOfCape` and `foldBack` function.

Performance: Fold vs. FoldBack

Which one is more efficient? and Why?

Problem 2: Reduce

Implement the `reduce` function. Raise an exception (with `failwith`) if the given list is empty.

Built-in Higher-Order Functions

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`
- exists: `('T -> bool) -> 'T list -> bool`
- And many more.

These functions take in an action (a function) as input.

Built-in Support!

All the previous 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.

Example Usage of Map and Reduce

Given a list of integers, return a sum of squares of all the elements.

```
let lst = [ 1; 2; 3; 4; 5 ]
lst
|> List.map (fun x -> x * x)
|> List.reduce (+)
```

Example Usage of Filter

Given a list of strings, return a list of strings that contains only strings whose length is greater than 3.

```
let lst = [ "a"; "ab"; "abc"; "abcd"; "abcde" ]
lst
|> List.filter (fun s -> String.length s > 3)
```

Higher-Order Functions for String

- `String.map: (char -> char) -> string -> string`
- `String.filter: (char -> bool) -> string -> string`
- `String.forall: (char -> bool) -> string -> bool`
- etc.

Example Usage of `String.map`

Given a string, return a string that contains only lower-case characters.

```
let str = "Hello, World!"
str
|> String.map (fun c -> if Char.IsUpper c then Char.
   ToLower c else c)
```

Example Usage of String and List

Given a list of strings, return a list of strings that contains only lower-case characters.

```
let lst = [ "Hello"; "World"; "abc" ]
lst
|> List.filter (String.forall Char.IsLower)
```

Set

Mathematical Set

A set is a collection of ***distinct*** objects.

Standard Set Operations

- Union ($A \cup B$).
- Intersection ($A \cap B$).
- Difference ($A \setminus B$).

Implement your own set data type

Now that we know the concept of ***data abstraction***, let's first pretend that there is a `Set<'T>` data type and we have three functions that operate on it.

```
val union: Set<'T> -> Set<'T> -> Set<'T>
val inter: Set<'T> -> Set<'T> -> Set<'T>
val minus: Set<'T> -> Set<'T> -> Set<'T>
```

Implementing Set with List

We can use List to implement Set!

```
type Set<'T> = List<'T>
val union: Set<'T> -> Set<'T> -> Set<'T>
val inter: Set<'T> -> Set<'T> -> Set<'T>
val minus: Set<'T> -> Set<'T> -> Set<'T>
```

❓ The expression "`type A = <type name>`" is called a "Type Abbreviation". You can give an alternative name for a type using this expression[1].

[1] `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/type-abbreviations`

Implementing Set with List

What's the problem?

Built-in Set

Implementing an efficient set data structure is beyond the scope of this course, but we will learn how to use built-in data structures.

- `Set.ofList`
- `Set.empty`
- `Set.add`
- `Set.remove`
- `Set.union`
- `Set.intersect`
- `Set.difference`

Map and Fold

Built-in Set stores its elements in an order, and the mapping and folding will happen in the order of the elements. Elements in Set should have a comparable types[2].

- `Set.map: ('T -> 'U) -> Set<'T> -> Set<'U>`
- `Set.fold: ('State -> 'T -> 'State) -> 'State -> Set<'T> -> 'State`

[2] This is so-called type constraints in F# jargon.

Map

Map?

A map is a data structure composed of a collection of (key, value) pairs, such that each possible key appears at most once in the collection.

Key	Value
k1	`"Beer"`
k2	`"Juice"`
k3	`"Soda"`

Standard Map Operations

- **Add**: add a key-value pair to the map.
- **Find**: find a value associated with a particular key.

Built-in Map

- `Map.ofList`
- `Map.empty`
- `Map.add`
- `Map.find`
- `Map.tryFind`
- `Map.containsKey`

Map and Fold

Built-in Map stores its elements in an order, and the mapping and folding will happen in the order of the elements.

- `Map.map: ('K -> 'V -> 'U) -> Map<'K, 'V> -> Map<'K, 'U>`
- `Map.fold: ('State -> 'K -> 'V -> 'State) -> 'State -> Map<'K, 'V> -> 'State`

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?