Midterm 2: A Brief Overview of Swift

by Group 8

Tong Zhao, Xiaowei Yang, Yun-Chien Wu, Stephen Trotter, Jonathan Toombs,
Mathew Sedgewick, Austin Hendy, Adar Guy, Marin Evergreen

Q1: Mathew Sedgwick, Xiaowei Yang
Q2: Jonathan Toombs, Austin Hendy
Q3: Adar Guy, Marin Evergreen
Q4:Tong Zhao, Yun-Chien Wu
Editor: Stephen Trotter

Question 1: Currying and Partial Function Application

Currying is defined as "the process of decomposing a function of multiple arguments into a chained sequence of functions of one argument"[1Q1] and is found in both Haskell and Swift. The main theoretical advantage to use a curried function is that "formal proofs are easier when all functions are treated uniformly"[2Q1]. Partially applied functions are functions that haven't been given all of the required parameters. In this case, the return type of the partial function becomes a new function that will take in the rest of the required parameters and return the originally intended value. Partial function applications actually go hand-in-hand with currying. For example, by using currying on a function that takes in two parameters and reducing it down to a function that only takes one, the "new" partial function then performs only part of its original functionality and returns another function that will require the other parameter to fulfill the rest of it.

Currying and partial function application allows user to create specialized functions based on general functions without introducing new code and repeating the old ones. For example, the following code retrieved from Gordon Fontenot's Introduction to Function Currying in Swift[3Q1], is a simple function that takes two integers (a and b) as arguments and then computes their sum. We are going to transform this function into a function that only takes one parameter by using currying, as shown.

```
func add(a: Int, b: Int) -> Int{
        return a + b
}
```

When we want to add a constant value to the function, in this case the number 2, we could create a quick list of numbers using range and then use 'Rang.map' to iterate through the values of a list and produce a new list where all previous values have been incremented by 2[3Q1].

```
func addTwo(a: Int) -> Int {
        return add(a, 2)
}
let xs = 1...100
let x = xs.maps(addTwo)    // x = [3, 4, 5, 6, etc]
```

A closure defines the environment of a function and inside this environment, there exists at least one bound variable[4Q1]. We do not need to use a closure for map if we have a function that takes one argument and return one value[3Q1]. In this case, we can pass the function to map directly by wrapping the 'add' function in a new function[3Q1]. However, to be more efficient, we can make alterations to the original function so that it only takes in one parameter[3Q1]:

```
func add(a: Int) -> (Int -> Int) {
        return { b in a + b }
}
```

Now, this function takes one single argument and returns a function[3Q1]. The returned function takes a single argument and returns the sum of a (passed into the ‘add’ function), and b (passed in by the closure)[3Q1]. This simply means that when we want to call it immediately and satisfy all arguments, we need to separate the arguments with parentheses like so[3Q1]:

```
let  sum = add(2)(3) // sum = 5
```

What the previous code depicts is a prime example of partial function applications. The add function takes in the value 2 and then returns an unnamed function that both takes in and returns an integer. The new, unnamed function then takes in the value 3, and which allows for the rest of the functionality to be completed and for the value 5 (the sum) to be returned.

By reducing the number of arguments needed, we can also create smaller functions out of larger functions. In this case, we can use “let” to define the our “addTwo” function as follows[3Q1]:

```
let  addTwo = add(2)
let  xs = 1,2, ...99,100
let  x = xs.map(addTwo)     // x = [3, 4, 5, 6, etc]
```

In our use, the ‘addTwo’ function takes a single int argument and returns an int, which will be passed directly to map[3Q1]. The map function simply applies a function to every item of a list. At this point, we create and return the closure and give our function an unintuitive signature[3Q1]. We are able to change the original add function to achieve the same currying functionality as the following[3Q1]:

```
func add(a: Int)(b: Int) -> Int {
        return a + b
}
```

Finally, the following code accomplishes the desired functionality and partially applies the function to map[3Q1].

```
func add(a: Int)(b: Int) -> Int {
        return a +b
}
let  addTwo = add(2)
let  xs = 1...100
let  x = xs.map(addTwo)     // x = [3, 4, 5, 6, etc]
```

The fundamental difference between Swift and Haskell, with respect to currying, is that Haskell’s functions are always curried, while Swift makes function currying available but leaves its’ use up to the developer[5Q1]. Swift can also accomplish currying applications with far less work by using anonymous variables[6Q1]. For example, Figures 1[6Q1] and 2[6Q1] depict the use of currying by adding the constant value ‘2’ to a list of numbers, both with and without anonymous variables, respectively. Swift, however, does not allow infix operators to be curried, unlike Haskell, which is able to apply fmap (2+) to a list/array[6Q1].

```
println([1, 2, 3, 4].map{return $0 + 2})
```

Figure 1.1 “Using anonymous variables to add 2 to each value in a list”, written in Swift

```
func addTwoNumbers(x : Int)(y: Int) -> Int {
    return x + y
}
let add2 = addTwoNumbers(2)
[1, 2, 3, 4].map{add2(y:$0)}
```

Figure 1.2 “Adds 2 to every value of a list without anonymous variables”, written in Swift

The implementation of Swift, unlike Haskell, allows function parameters to have default values that are specified in the function declaration[5Q1]. A default parameter value is a compile-time binding between a parameter and a constant value and cannot be reassigned a new value at run-time. Currying, however, allows you to work around this and effectively allows for a parameter’s default value to be determined at run-time[5Q1]. Partially applying a function results in a new function that knows the value of the bound parameter[5Q1]. “Unlike using a default parameter, code which calls that new function cannot specify a different value for the bound parameter. Once a value has been bound via the partial function application, that parameter value is set in stone”[5Q1].

Question 2: Lazy Evaluation and Infinite Lists

In the following section we will discuss Swift's lazy evaluation, also known as call-by-need, as well as its’ effect when used in conjunction with infinite lists.

Lazy evaluation is a powerful tool when used properly. Memory is an important resource in all applications and should be optimized whenever possible. Lazy evaluation provides the ability to ensure that the actions required to retrieve a value by a generator is only performed when necessary [1Q2]. In short, if the program has found what is needed it does not need to perform excess iterations in order to complete the process.

Lazy evaluation and lazy initialization have very similar purposes. They both strive to reduce computation time, and only perform their tasks when necessary. An

interesting lazy initialization rule includes the use of “var” instead of “let”. This is because let is used for constants, which have the property of having a value before initialization is completed [2Q2]. In the case of lazy initialization it may be the case that the initial value may not be retrieved until after instance initialization completes. [2Q3]

```
func generateRandomNumber()->Int{
   print("calculating")
  return random()
}

class roomate {
   var name : String = "Austin Hendy"
   var age = 20
   lazy var favouriteNumber = generateRandomNumber()
}

func giveInfo() -> String{
   return("\(name) is \(age) years old")
}

var newRoomate = roomate()
```

fig. 2.1: Swift Lazy Example

As one may observe in figure 2.1, when newRoomate is called it will trigger the name, age, and favouriteNumber. Although there is no reason for favouriteNumber to be called as even if the giveInfo() function is called it is still not used. This is easily fixed by adding the lazy attribute in front of the statement like so:

```
lazy var favouriteNumber = generateRandomNumber();
```

with this it eliminates the need to call the extra function unless it is accessed with newRoomate.favouriteNumber.

Lazy evaluation saves time when determining the largest element after quick sorting. If one were to calculate the max of the list [8,10,4,7,6]; with the pivot being position 0 (value 8), it would then grab everything bigger than 8 which is just 10. As it is recursive it would call it again returning 10 as its the only bigger value in the list. If lazy evaluation were to be used, the “less than” subarray would have never been sorted saving computation time.

Even infinite data structures can be handled effectively with lazy evaluation. Infinite lists are usually not considered when writing code, but are possible in functional languages such as Haskell and Swift. Swift builds infinite lists using a type called GeneratorTypes, which are used to build Generators. Generators have a single instance method: next(), which calculates and returns the next value.

When looking at infinite data structures like an array, it is not evaluated at all until a specific value (or index) needed, and only until it reaches the data that is needed. For example, if one had an array of size infinity and what was needed was in index 2; with lazy evaluation it would only perform 3 iterations through the array leaving the rest of the array unevaluated.

A great application of lazy evaluation is on fibonacci numbers[2Q6]. The code is found in appendix 2.1.

Figure 2.2 Describes how to create an infinite sequence and lazily evaluate until a flag is set.

```
let seq = GeneratorSequence(FibonacciGenerator())
for n in seq {
    print("\(n), ")
    if (n > 100000) {
        break;
    }

}
```

fig 2.2: An Infinite Sequence Generator

This code will stop after printing the first Fibonacci number greater than 100000, but this code could be changed to any value and it would work. For example: n> 100,000 will return 121393. n>1,000,000 returns 1346269.

If you have built an infinite sequence you can turn this sequence type into a LazySequenceType by placing .lazy after the sequence. Once the sequence is lazy, commands, such as map, reduce, and filter, can be used on the sequence on demand (lazily, through next()).
Figure 2.3 describes swift code to create, filter and lazily evaluate an infinite list. [5Q2]

```
class InfiniteSequence :SequenceType {
    func generate() -> AnyGenerator<Int> {
```

```
        var i = 0
        return anyGenerator({
            return i++
        })
    }
}
var fs = InfiniteSequence().lazy.filter({$0 % 2 == 0}).generate()
for f in fs.prefix(20){
    print(f)
}
> 0, 2, 4. ... , 38
```

fig. 2.3: An infinite Sequence that uses lazy evaluation

Laziness allows you to filter the Infinite Sequence into the first 20 odd numbers. A sequence without prefix(n) would run until memory has run out. [5Q2]

There are some downsides to lazy evaluation as they can be a hassle to combine with I/O and exception handling as the order of operations becomes inconclusive [2Q4]. Careless use of lazy evaluation can also easily lead to memory leaks. Runtime can also become very hard to predict as do not know how long it will take to example find the nth element in an infinite array.

Question 3: Higher Order Functions and Closure

Closure

In programming closures are techniques for the implementation of lexically scoped name bindings in languages with first-class functions. If a language has first-class functions, then the language supports passing functions as arguments to other functions, returning functions as values from other functions, and assigning them to variables or storing them in data structures [3Q3]. Closure in this context is a set of records that stores a function. It maps each free variable (variables used locally, but defined in enclosing scope) to a storage location or value that it was bound to when created. Closures are essentially functions that refer to independent (free) variables. The function defined in the closure can be said to “remembers” the environment in which it was created. Unlike a plain function, closure allows the function to access those stored variables through the reference that the closure has created, even when the function is called outside the scope of those variables [1Q3].

In the following example, the function “nestedFunc()” with parameter “b” is nested within the high-order function “highOrder()” with parameter “a”:

```
function highOrder(a)
        function nestedFunc(b)
                return a + b
        return nestedFunc
variable closureOne = highOrder(6)
variable closureTwo = highOrder(11)
```

The nested function has access to the variable “a” because it is within the lexical scope of a, even though a is not local in the function “nestedFunc()”. It is the high-order function that returns a closure containing the nested function which executes the calculation. The variables “closureOne” and “closureTwo”, being of function types can also be called. closureOne(2) will return 8 and closureTwo(4) will return 15 showing that although they refer to the same function, the associated environments differ and invoking each function type will bind the variable name “a” to two distinct variables with different values in the two invocations, which returns two different results.

Any functional programming language must support higher-order functions and implement closure correctly. Some programming languages can function as both practical and functional languages, and have masterfully created a platform that can include closure and higher-order functions while supporting plain functions in the same environment. JavaScript is an example of a practical programming language that supports both concepts. In JavaScript, a closure can be understood in two different ways; *a local variable for a function that is kept alive after the function has returned* or *a stack-frame which is not de-allocated when the function returns (if a stack-frame was malloc’ed instead of being on the stack)* [2Q3].

```
function say667() {
        var num = 666;
        var say = function() { console.log(num); }
        num++;
        return say;
}
var sayNumber = say667();
sayNumber(); // logs 667
```

This example shows that the local variables are kept by reference kind of like keeping a stack-frame in memory when the outer function exists.

In many other languages, the idea of function-to-variable assignment can be thought of simply as returning a pointer to a function, and that the variable name (“x” in var x) is a pointer to the function which it is being assigned to [6Q3]. But JavaScript has a unique approach when creating a reference to a function. When a variable is assigned to a function in JavaScript, a pointer to a function is created, as

well as a hidden pointer to a closure [2Q3]. While most languages destroy the stack-frame making all local variables no longer accessible, JavaScript maintains this closure pointer as a reference for outside the function [5Q3]. This means that when you declare a function within another function, the local variables can remain accessible after returning from the function you called. This is demonstrated in the example above, because the function sayNumber() is called after the function say667() is returned. The call references the variable “num”, which was a local variable of the function say667(). Although this is not a traditional implementation of closure, it provides JavaScript with the platform to be both practical and functional as a language.

Swift contains three pillars which allow it to be functional as a language, and implement closure efficiently. The first pillar is that Swift supports first-class variables, so functions can be assigned to variables, and passed through as parameters into other functions and so on [8Q3]. This allows a programmer to easily construct higher-order functions and apply them in useful ways. The member function *map* for instance takes a function as a parameter and applies it to each member of the array, returning a new array that contains the results.

```
func doubler(i : Int) -> Int { return i * 2 } // function that doubles an Int
let a = [1, 2, 3, 4].map(doubler) // a contains [2, 4, 6, 8]
```

The next pillar is that functions can also “capture” or use variables that exist outside of their local scope (as long as it exists within the lexical scope of the higher-order function) and the last pillar is that there are two ways of implementing a function in Swift; with the *func* keyword or with { } (although Swift confusingly calls the latter “closure expressions” which can be confused with closure). Below is an example of the doubler function from above, written using the closure expression syntax:

```
Let doubler = { (i : Int) -> Int in return i * 2 }
[1, 2, 3].map(doubler)
```

The doubler examples are absolutely equivalent and even exist in the same “namespace” unlike in some other languages, but the second example can be thought of as a function literal like “1” for ints or “hello” for strings [4Q3]. Unlike the func keyword, they are unnamed and must assigned to a variable. Understanding these 3 facts about functions and closure in Swift is crucial to the ability to use and take full advantage of the implementations in Swift.

When comparing JavaScript and Swift’s closure implementations, some interesting points can be made. One of the most notable differences involves the

third pillar of Swift's closure implementation. Because there are two ways to implement a function, Swift closures do not require a keyword (such as *func*) and so the implementation is more concise, this is contrasted by JavaScript, which can be quite verbose [7Q3]. Because of the difference in scoping, Swift also does not suffer from the same problem of using closures within loops. This is because a JavaScript implementation of a loop is the traditional one that re-uses the variable, but in Swift the *for…in* loop makes a new iteration of the variable each time.

```
var funcs = [];
function createFunc(i) {
        return function() { console.log("My value:" + i };
}
for (var i = 0; i < 3; i++) {
        funcs[i] = createfunc(i);
}
for (var j = 0; j < 3; j++) {
        funcs[j]();
}
```

JavaScript Implementation

```
var funcs : [( ) -> Int] = []
for i in 0 … 3 {
        funcs.append ({
                return i
        })
}
for i in 0 … 3 {
        println(funcs[i]( ))
}
```

Swift Implementation

This example on the left is a piece of JavaScript code, which returns the value defined in the loop. The variable is bound within each function to a separate, unchanging value outside the function. Since there is no block scope in JavaScript (only function scope), wrapping the function creation in a new function ensures that the value of "i" remains as intended [5Q3]. It is quite apparent that the case is not the same with Swift's implementation of the same program (right). There is no need to wrap the *funcs* function in an outer function because the variable "i" is always reiterated in every execution of the loop.

```
let newFunction =  oldFunction()
newFunction()
// OR
var newFunction = oldFunction()
```

newFunction()

JavaScript and Swift Implementation

Both JavaScript and Swift implement first-class functions in the same manner. They each provide a platform for which to pass functions as parameters, assign them to variables, and return them or store them in data structures. Even the syntax is similar. Both can use *let* to set an immutable variable to a function (the *var* keyword has function scope but not block scope whereas the *let* keyword has local scope) [2Q3]. Although, if a function is implemented in Swift using closure expressions, then it will require assignment before being used as a parameter or return value. Lastly, the implementation of variable “capturing” outside the local scope is implemented quite similarly for both Swift and JavaScript [4Q3]. Swift creates capture lists that appear as the first item in a closure. They are listed before any parameter or identifier lists, and require the in keyword to be used and require the in keyword to be used. JavaScript similarly creates capture lists in a closure but captures only variables, not values[1Q3]. The *var* keyword has function scope but not block scope. Regardless of the syntactical differences in the languages (which are few), both implement closure in similar ways, with Swift taking the highlights from closures in JavaScript, and combining them with the structures of blocks in C and Objective-C. Both implement closure in relatively efficient and appropriate manners[6Q3].

Higher-Order Functions

Higher-order functions are a powerful component of functional languages. They are described as functions that either accept a function as an argument or return a function [9Q3]. Some examples of higher-order functions are function composition, fold/reduce, map, and filter; all of which accept or return a function. Function composition consists of building functions out of other functions; that is, evaluating a function, and using the result as the parameter for the next function [10Q3]. The fold function, also known as reduce, accepts a function and a data structure, then combines the elements of the data structure in some way to generate a return value [11Q3]. The map function accepts a function and a list, and generates a new list by applying the function to each element in the list [12Q3]. Filter employs a predicate, which is a function that returns a boolean value, by applying it to elements of a data structure [13Q3]. It returns the elements of the data structure that satisfy the predicate. According to Haskell documentation, a reason for using higher-order functions is to “abstract common behaviour into one place”, allowing the programmer to reduce redundancy in code [14Q3].

Functions are considered first class citizens in Swift, which means that variables can be associated to functions, and that functions can be passed around [15Q3].

This trait allows for support of higher-order functions in the language, since it means that functions can be passed as arguments and returned from other functions. The language has built-in support for most of the higher-order functions listed above: reduce, map, and filter. Function composition is easily defined in the language by the programmer. The following example demonstrates this [16Q3]:

```
func parseUsers(dictionaries: [[String: AnyObject]]) -> [User] {
        return dictionaries.map({ dict in
                return User(dictionary: dict)
        })
}
```

The above example takes in an array of user dictionaries as input and transforms them into their object representation. It is an example of both the map function and function composition.

Functions are also considered first class citizens in JavaScript, allowing the language to implement functional behaviour [17Q3]. An example of taking a function as an argument is a function that uses a callback. A callback is a function that is generally passed as the last argument to a function. It is executed after everything else has finished executing, and it allows the program to behave asynchronously. JavaScript also allows functions to be returned. As a result, JavaScript offers support for higher-order functions. It also offers built-in functions for reduce, map, and filter. It is simple to build function composition, as demonstrated by the following code snippet, which also demonstrates the reduce and filter functions [18Q3]:

```
function average(array) {
        function plus(a, b) { return a + b; }
        return array.reduce(plus) / array.length;
}
function age(p) { return p.died – p.born; }
function male(p) { return p.sex == “m”; }
function female(p) { return p.sex == “f”; }

console.log(average(ancestry.filter(male).map(age)));
console.log(average(ancestry.filter(female).map(age)));
```

The above example defines plus as a function since operators cannot be passed as arguments. It first filters the ancestry array based on gender, then maps to age. The average is calculated on the array by calling reduce with the plus operator.

Swift and JavaScript are similar in that the two languages allow accepting functions as parameters and returning functions, and in that they both consider their function first class. One difference is that Swift is strongly-typed. Like any other type,

functions have a type, which differentiates it from purely functional languages [19Q3]. In JavaScript, which is not strongly-typed, almost everything is an object except for primitive values [20Q3]. This means that functions are function objects [21Q3]. Despite this difference, both languages effectively and efficiently support the powerful concept of higher-order functions.

Question 4: List Comprehensions, Map, Filter

Many programing languages, such as Python and Haskell, support a concept called "list comprehensions". List Comprehensions are a powerful tool, which allows the creation of a new list based on another list, in a single, readable line. Using list comprehensions can help programmers code more efficiently and increase readability.

Common applications for list comprehensions are making new lists where each element is the result of some operations applied to each member of another list. Or to create a sub-sequence of those elements that satisfy a certain condition.

This is an annotated example:

$$S = \{ \underbrace{2 \cdot x}_{\text{output expression}} \mid \underbrace{x}_{\text{variable}} \in \underbrace{\mathbb{N}}_{\text{input set}} , \underbrace{x^2 > 3}_{\text{predicate}} \}$$ [1Q4]

A list comprehension has the same syntactic components to represent generation of a list in order from an input list or iterator:

- An output expression which produces each member of the output list from members of the input list and satisfy the predicate.
- A variable to represent members of the input list.
- An input list.
- An optional predicate expression.

List comprehensions cannot be used when the construction rule is too complicated to be expressed with "for" and "if" statements. If this is the case, other tools such as map or filter may be used with an appropriate function. These can even be combined that with list comprehensions for greater flexibility. Fortunately, Swift has function map and filter so that the programmer can accomplish list comprehension-like operations in Swift even though Swift does not provide an explicit syntax for list (or array) comprehension[2Q4].

Swift's standard Array Library includes support for 3 higher order sequence functions: map, filter and reduce[3Q4]. Let us look more in depth at the map and filter methods.

The map method solves the problem of transforming the elements of an array using a function by apply the function to each element of the list.

```
[x1, x2, ... , xn].map(f) => [f(x1), f(x2), ... , f(xn)]
```

or

```
map((x1..xn), f)
```

Map is declared as a method on the Array class with signature

```
func map<U>(transform: (T) -> U) -> U[]
```

[3Q4].

That means that it receives a function named transform that maps the array element type T to a new type U and returns an array of U.

The filter method solves the problem of selecting the elements of an array that pass a certain condition. Filter can be called in two different ways:

```
array.filter(pred)
```

or

```
filter(sequence, pred)
```

Filter is declared as a method on the Array class with signature:

```
func filter(includeElement: (T) -> Bool) -> T[]
```

[3Q4].

It receives a function includeElement that returns true or false for elements of the array and returns only the elements that return true when includeElement is called on them. With map and filter methods, the concept of list comprehension can be implemented in Swift. For example, if the programmer would like to create a list of squares:

In Haskell,

```
squares = [x*x | x <- [1..10]]
```

In Swift,

```
let squares = Array(map(1..10) {$0 * $0})
```

Another example with listing the odd numbers:

In Haskell,

```
odd = [x | x <- [1..10], odd x]
```

In Swift,

```
let odd = (1..10).filter {$0 % 2 !=0}
```

As we can see, list comprehensions, map and filter are powerful tools for any programmer.

Appendix:

Q2: Fibonacci Sequence Generator.

```
class FibonacciGenerator: Generator {
    var n1 = 0
    var n2 = 0

    func next() -> Int? {
        switch (n1, n2) {
        case (0, 0):
            n2 = 1
            return 1
        case (0, 1):
            n1 = 1
            return 1
        default:
            let result = n1 + n2
            n1 = n2
            n2 = result
            return result
        }
    }
}
```

References:

Q1:
[1Q1]
http://www.vasinov.com/blog/on-currying-and-partial-function-application/#toc-pfa
[2Q1] https://wiki.haskell.org/Currying
[3Q1] https://robots.thoughtbot.com/introduction-to-function-currying-in-swift
[4Q1] https://simple.wikipedia.org/wiki/Closure_(computer_science)
[5Q1] http://ijoshsmith.com/2014/06/09/curried-functions-in-swift/

[6Q1]
http://ericasadun.com/2015/05/14/swift-curry-take-out-with-a-side-of-image-processing/

Q2:
[1Q2] http://blog.scottlogic.com/2014/06/26/swift-sequences.html
[2Q2] http://mikebuss.com/2014/06/22/lazy-initialization-swift/
[3Q2] http://stackoverflow.com/questions/27886024/lazy-var-vs-let
[4Q2]
https://en.wikipedia.org/wiki/Lazy_evaluation#Working_with_infinite_data_structures
[5Q2]
https://www.uraimo.com/2015/11/12/experimenting-with-swift-2-sequencetype-generatortype/
[6Q2] https://gist.github.com/kristopherjohnson/9b64cb05fe764211728c

Q3:
[1Q3] http://jamesonquave.com/blog/functional-programming-in-swift/
[2Q3]
http://www.raywenderlich.com/114456/introduction-functional-programming-swift
[3Q3] http://five.agency/functional-programming-in-swift/
[4Q3] https://developer.mozilla.org/en/docs/Web/JavaScript/Closures
[5Q3] https://en.wikipedia.org/wiki/Closure_%28computer_programming%29
[6Q3] https://reprog.wordpress.com/2010/02/27/closures-finally-explained/
[7Q3] http://robnapier.net/swift-is-not-functional
[8Q3]
https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Closures.html
[9Q3] http://homepage.cs.uiowa.edu/~slonnegr/plf/Book/Chapter5.pdf
[10Q3] https://en.wikipedia.org/wiki/Function_composition_(computer_science)
[11Q3] https://en.wikipedia.org/wiki/Fold_(higher-order_function)
[12Q3] https://en.wikipedia.org/wiki/Map_(higher-order_function)
[13Q3] https://en.wikipedia.org/wiki/Filter_(higher-order_function)
[14Q3] https://wiki.haskell.org/Higher_order_function
[15Q3] http://blog.tackmobile.com/article/functional-paradigms-in-swift/
[16Q3] http://nsexceptional.com/functional-swift-higher-order-functions/
[17Q3] http://www.sitepoint.com/higher-order-functions-javascript/
[18Q3] http://eloquentjavascript.net/05_higher_order.html
[19Q3]
https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Functions.html

[20Q3]
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function
[21Q3] http://www.w3schools.com/js/js_object_definition.asp

Q4:
[1Q4] https://en.wikipedia.org/wiki/List_comprehension/
[2Q4]
http://jamesonquave.com/blog/list-comprehensions-and-performance-with-swift/
[3Q4]
https://www.weheartswift.com/higher-order-functions-map-filter-reduce-and-more/