Function Programming in Haskell (CS 430 notes)

"Writing functional code doesn't require a shift to a functional programming language such as Scala or Clojure [or Haskell] but rather a shift in the way you approach problems." - Neal Ford (Functional Thinking, 2014)

"Let's say for a moment that you are a lumberjack. You have the best axe in the forest, which makes you the most productive lumberjack in the camp. Then one day someone shows up and extols the virtues of a new tree-cutting paradigm, the chainsaw. [...] Demonstrating your expertise with the previous tree-cutting paradigm, you swing it vigorously at a tree—without cranking it. You quickly conclude that this newfangled chainsaw is a fad, and you return to your axe. Then, someone appears and shows you how to crank the chainsaw. **The problem with a completely new programming paradigm isn't learning a new language. [...] The tricky part is learning to think in a different way.**" - Neal Ford (Functional Thinking, 2014)

"Once garbage collection became mainstream, it simultaneously eliminated entire categories of hard-to-debug problems and allowed the runtime to manage a process that is complex and error-prone for developers. Functional programming aims to do the same thing for the algorithms you write, allowing you to work at a higher level of abstraction while freeing the runtime to perform sophisticated optimizations." - Neal Ford (Functional Thinking, 2014)

Functional programming:

- "what" instead of "how"
- definitions instead of algorithms
- functions instead of statements
- recursion instead of iteration
- parameters instead of variables
- no state (or strictly managed state)

Haskell:

- strong, static typing w/ inference
- pure w/ no side effects
- first-order functions
- built-in lists w/ comprehensions
- filter, map, fold/reduce
- pattern matching
- currying and partial application
- non-strict w/ lazy evaluation
- monads for state

```
add x y = x + y
add3 = add 3
add3 7
```

```
:t head
:t tail
head [1,2,3]
tail [1,2,3]

[2..]
take 5 [2..]
```

Example 2-1. Typical company process (in Java)

```java
package com.nealford.functionalthinking.trans;

import java.util.List;

public class TheCompanyProcess {
    public String cleanNames(List<String> listOfNames) {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < listOfNames.size(); i++) {
            if (listOfNames.get(i).length() > 1) {
                result.append(capitalizeString(listOfNames.get(i))).append(",");
            }
        }
        return result.substring(0, result.length() - 1).toString();
    }

    public String capitalizeString(String s) {
        return s.substring(0, 1).toUpperCase() + s.substring(1, s.length());
    }
}
```

Example 2-3. Processing functionally in Scala

```scala
val employees = List("neal", "s", "stu", "j", "rich", "bob", "aiden", "j", "ethan",
         "liam", "mason", "noah", "lucas", "jacob", "jayden", "jack")

val result = employees
  .filter(_.length() > 1)
  .map(_.capitalize)
  .reduce(_ + "," + _)
```

Example 2-4. Java 8 version of the Company Process

```
public String cleanNames(List<String> names) {
    if (names == null) return "";
    return names
            .stream()
            .filter(name -> name.length() > 1)
            .map(name -> capitalize(name))
            .collect(Collectors.joining(","));

}

private String capitalize(String e) {
    return e.substring(0, 1).toUpperCase() + e.substring(1, e.length());
}
```

In the Scala version, I can make the code parallel by adding `par` to the stream, as shown in Example 2-8.

Example 2-8. Scala processing in parallel

```
val parallelResult = employees
  .par
  .filter(_.length() > 1)
  .map(_.capitalize)
  .reduce(_ + "," + _)
```

I can make an almost identical change to the Java 8 version to achieve the same effect, as shown in Example 2-9.

Example 2-9. Java 8 parallel processing

```
public String cleanNamesP(List<String> names) {
    if (names == null) return "";
    return names
            .parallelStream()
            .filter(n -> n.length() > 1)
            .map(e -> capitalize(e))
            .collect(Collectors.joining(","));
}
```

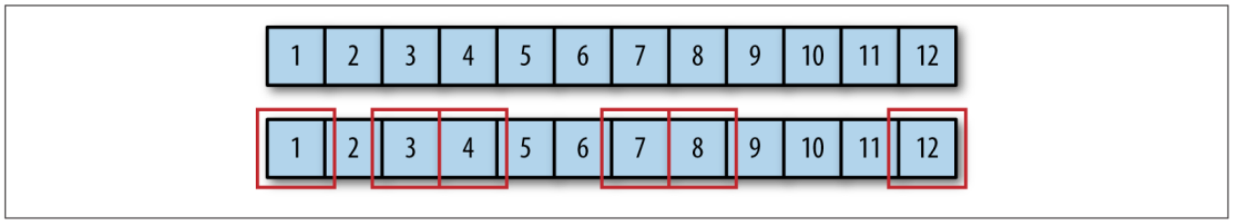

Figure 2-1. Filtering a list of numbers from a larger list

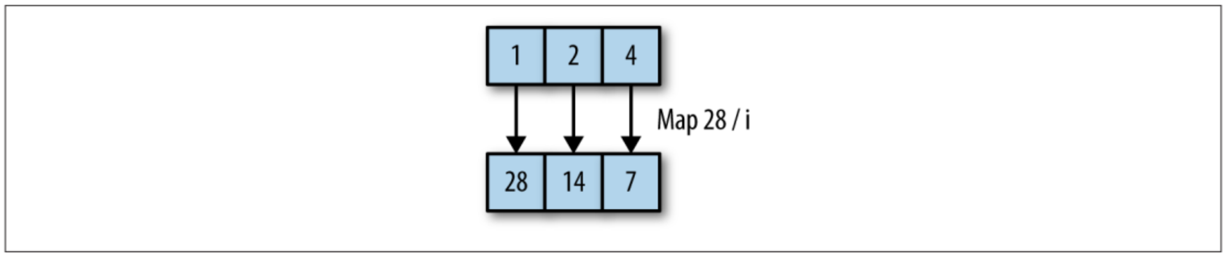

Figure 2-2. Mapping a function onto a collection

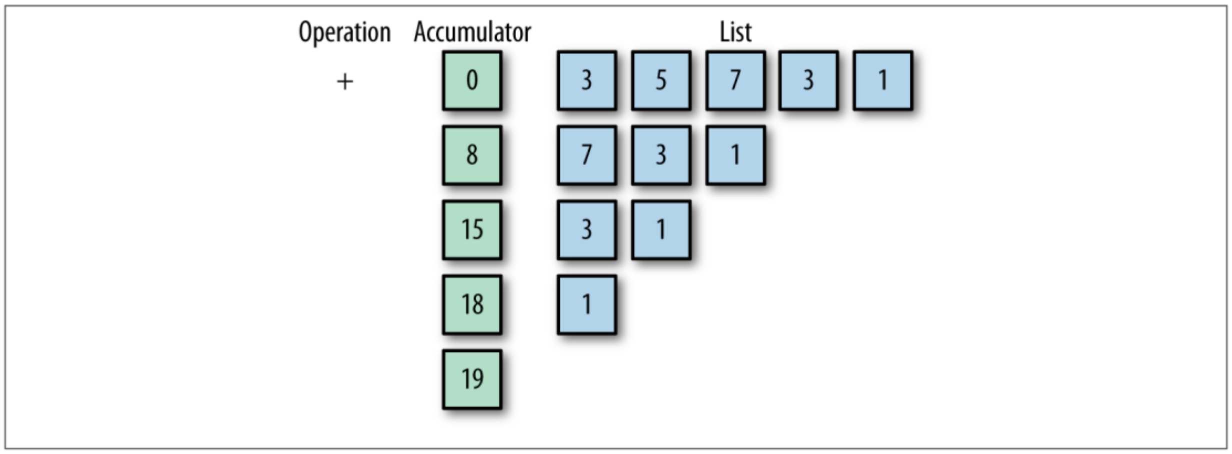

Figure 2-3. Fold operation

```
-- cat utility one-liner, courtesy of @cattheory on Twitter
--
main = getArgs >>= mapM_ (readFile >=> putStrLn)
```

References:

- CPL Ch.15
- https://www.safaribooksonline.com/library/view/functional-thinking/9781449365509/
- https://www.safaribooksonline.com/library/view/real-world-haskell/9780596155339/