FREE eBook

LEARNING functional-programming

Free unaffiliated eBook created from **Stack Overflow contributors.**

#functional-programming

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: functional-programming

It is an unofficial and free functional-programming ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official functional-programming.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with functional-programming

Remarks

Functional programming is a programming paradigm which models computations (and thus programs) as the evaluation of mathematical functions. It has its roots in lambda calculus, which was developed by Alonzo Church in his research on computability.

Functional programming has some interesting concepts:

- **Higher Order functions**
- **Purity**
- **Recursion**
- **Laziness**
- **Referential Transparency**
- **Currying**
- **Functors**
- **Monads**
- **Memoization & Tail-call Optimization**
- **Functional Unit Testing**

Some examples of functional programming languages are Lisp, Haskell, Scala and Clojure, but also other languages, like Python, R and Javascript allow to write (parts of) your programs in a functional style. Even in Java, functional programming has found its place with Lambda Expressions and the Stream API which were introduced in Java 8.

Examples

Pure functions

Pure functions are self-contained, and have no side effects. Given the same set of inputs, a pure function will always return the same output value.

The following function is pure:

```
function pure(data) {
    return data.total + 3;
}
```

However, this function is not pure as it modifies an external variable:

```
function impure(data) {
    data.total += 3;
    return data.total;
```

```
}
```

Example:

```
data = {
    total: 6
};

pure(data);   // outputs: 9
impure(data); // outputs: 9 (but now data.total has changed)
impure(data); // outputs: 12
```

Higher-order functions

Higher-order functions take other functions as arguments and/or return them as results. They form the building blocks of functional programming. Most functional languages have some form of filter function, for example. This is a higher-order function, taking a list and a predicate (function that returns true or false) as arguments.

Functions that do neither of these are often referred to as `first-order functions`.

```
function validate(number,predicate) {
    if (predicate) {    // Is Predicate defined
        return predicate(number);
    }
    return false;
}
```

Here "predicate" is a function that will test for some condition involving its arguments and return true or false.

An example call for the above is:

```
validate(someNumber, function(arg) {
    return arg % 10 == 0;
    }
);
```

A common requirement is to add numbers within a range. By using higher-order functions we can extend this basic capability, applying a transformation function on each number before including it in the sum.

You want to add all integers within a given range (using Scala)

```
def sumOfInts(a: Int, b: Int): Int = {
  if(a > b) 0
  else a + sumOfInts(a+1, b)
}
```

You want to add squares of all integers within a given range

```
def square(a: Int): Int = a * a

def sumOfSquares(a: Int, b: Int): Int = {
  if(a > b) 0
  else square(a) + sumOfSquares(a + 1, b)
}
```

Notice these things have 1 thing in common, that you want to apply a function on each argument and then add them.

Lets create a higher-order function to do both:

```
def sumHOF(f: Int => Int, a: Int, b: Int): Int = {
  if(a > b) 0
  else f(a) + sumHOF(f, a + 1, b)
}
```

You can call it like this:

```
def identity(a: Int): Int = a

def square(a: Int): Int = a * a
```

Notice that `sumOfInts` and `sumOfSquare` can be defined as:

```
def sumOfInts(a: Int, b: Int): Int = sumHOF(identity, a, b)

def sumOfSquares(a: Int, b: Int): Int = sumHOF(square, a, b)
```

As you can see from this simple example, higher-order functions provide more generalized solutions and reducing code duplication.

> I have used Scala By Example - by Martin Odersky as a reference.

Currying

Currying is the process of transforming a function that takes multiple arguments into a sequence of functions that each has only a single parameter. Currying is related to, but not the same as, partial application.

Let's consider the following function in JavaScript:

```
var add = (x, y) => x + y
```

We can use the definition of currying to rewrite the add function:

```
var add = x => y => x + y
```

This new version takes a single parameter, `x`, and returns a function that takes a single parameter, `y`, which will ultimately return the result of adding `x` and `y`.

```
var add5 = add(5)
var fifteen = add5(10) // fifteen = 15
```

Another example is when we have the following functions that put brackets around strings:

```
var generalBracket = (prefix, str, suffix) => prefix + str + suffix
```

Now, every time we use `generalBracket` we have to pass in the brackets:

```
var bracketedJim = generalBracket("{", "Jim", "}") // "{Jim}"
var doubleBracketedJim = generalBracket("{{", "Jim", "}}") // "{{Jim}}"
```

Besides, if we pass in the strings that are not brackets, our function still return a wrong result. Let's fix that:

```
var generalBracket = (prefix, suffix) => str => prefix + str + suffix
var bracket = generalBracket("{", "}")
var doubleBracket = generalBracket("{{", "}}")
```

Notice that both `bracket` and `doubleBracket` are now functions waiting for their final parameter:

```
var bracketedJim = bracket("Jim") // "{Jim}"
var doubleBracketedJim = doubleBracket("Jim") // "{{Jim}}"
```

Immutability

In traditional object-oriented languages, `x = x + 1` is a simple and legal expression. But in Functional Programming, it's *illegal*.

Variables don't exist in Functional Programming. Stored values are still called variables only because of history. In fact, they are *constants*. Once `x` takes a value, it's that value for life.

So, if a *variable* is a *constant*, then how can we change its value?

Functional Programming deals with changes to values in a record by making a copy of the record with the values changed.

For example, instead of doing:

```
var numbers = [1, 2, 3];
numbers[0] += 1; // numbers = [2, 2, 3];
```

You do:

```
var numbers = [1, 2, 3];
var newNumbers = numbers.map(function(number) {
    if (numbers.indexOf(number) == 0)
        return number + 1
    return number
});
```

```
console.log(newNumbers) // prints [2, 2, 3]
```

And there are no loops in Functional Programming. We use recursion or higher-order functions like `map`, `filter` and `reduce` to avoid looping.

Let's create a simple loop in JavaScript:

```
var acc = 0;
for (var i = 1; i <= 10; ++i)
    acc += i;
console.log(acc); // prints 55
```

We can still do better by changing `acc`'s lifetime from global to local:

```
function sumRange(start, end, acc) {
    if (start > end)
        return acc;
    return sumRange(start + 1, end, acc + start)
}
console.log(sumRange(1, 10, 0)); // 55
```

No variables or loops mean simpler, safer and more readable code (especially when debugging or testing - you don't need to worry about the value of `x` after a number of statements, it will never change).

Read Getting started with functional-programming online: https://riptutorial.com/functional-programming/topic/3184/getting-started-with-functional-programming

Chapter 2: Loops by Recursive and Tail Recursive Functions

Introduction

As you already know, for the sake of immutability you can't process data using for loops and while loops. So we have recursive functions to rescue.

Examples

non-recursive (where immutability isn't a concern)

```
function sum(numbers) {
    var total = 0;
    for (var i = numbers.length - 1; i >= 0; i--) {
        total += numbers[i];
    }
    return total;
}
```

It's a procedural code with mutations (over `total`).

recursive to rescue

```
function sum(numbers) {
    if(numbers.length == 0) {
        return 0;
    }
    return numbers[0] + sum(numbers.slice(1));
}
```

this is the recursive version. there's no mutation, but we are making a call stack as below which uses extra memory.

sum([10, 5, 6, 7]);

10 + sum([5, 6, 7]);

10 + 5 + sum([6, 7]);

10 + 5 + 6 + sum([7]);

10 + 5 + 6 + 7 + sum([]);

10 + 5 + 6 + 7 + 0;

tail recursive to optimize

```
function sum(numbers) {
    return tail_sum(numbers, 0);
}

function tail_sum(numbers, acc) {
    if(numbers.length == 0) {
        return acc;
    }
    return tail_sum(numbers.slice(1), acc + numbers[0]);
}
```

in the tail recursive version, function return value does not need to wait till the end to do it its calculations, so there's no huge stack here; only two levels.

sum([10, 5, 6, 7]);

tail_sum([10, 5, 6, 7], 0);

tail_sum([5, 6, 7], 10);

tail_sum([6, 7], 15);

tail_sum([7], 21);

tail_sum([], 28);

28;

Read Loops by Recursive and Tail Recursive Functions online: https://riptutorial.com/functional-programming/topic/10819/loops-by-recursive-and-tail-recursive-functions

Credits

S. No	Chapters	Contributors
1	Getting started with functional-programming	AJW, Avinash Anand, Community, Huy Vo, Keelan, MauroPorrasP, Mr Tsjolder, rasmeister, rst_ack, tomturton, Zoyd
2	Loops by Recursive and Tail Recursive Functions	Dariush Alipour