Functional Programming in Ruby

Coding with Style

Koen Handekyn

Functional Programming in Ruby

Coding with Style

koen handekyn

This book is for sale at http://leanpub.com/functionalprogramminginruby

This version was published on 2014-04-17

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help koen handekyn by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought Functional Programming in Ruby

The suggested hashtag for this book is #fpir.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#fpir

Dedicated to my parents for the unconditional support to walk my own path.

Contents

‘For’ considered Harmful

In 1968 Dijkstra pointed this out: “Go To Considered Harmful” (paper[1]). He advocated structured programming instead, a programming paradigm aimed on improving the clarity, quality, and development time of a computer program by making extensive use of subroutines, block structures and for and while loops—in contrast to using simple tests and jumps.

In this book I’ll be advising you to move to an even more structured style of programming. I’ll try to convince you to write code without for and while loops. We will be replacing those coding structures with even higher level constructs.

article with comments to process http://david.tribble.com/text/goto.html

Diving in with an Example

Imagine you have a collection of references to stores. Each store is an object that has a name, a distance from where you are currently and the list price of a thing you want to buy.

This list of stores might have been loaded from a database, or a file, or received as JSON. It doesn’t really matter here. We also use plain hashes to represent the objects.

The challenge is to find the price of the cheapest provider that is within cycling distance. You are a sporty person and let’s say that is 5 miles. There’s also one extra thing to note: you have 10% discount on the list price at the Shell shops.

```
stores = []
stores << { name: "total", d: 2.0, price: 32.0 }
stores << { name: "shell", d: 2.6, price: 28.5 }
stores << { name: "esso", d: 3.2, price: 41.0 }
stores << { name: "q8", d: 3.5, price: 22.0 }
stores << { name: "shell", d: 4.5, price: 19.0 }
stores << { name: "q8", d: 5.5, price: 18.0 }
```

Comparing Styles

Do you prefer giving commands or instructions? Or are you the type that prefers explaining, empowering your people or colleagues? Maybe unexpected but this is exactly the difference between classic *imperative* programming styles and a more *functional* programming styles.

- Imperative = emphasizes on how something is computed.
- Imperative = make known or explain.

[1] http://www.u.arizona.edu/~rubinson/copyright_violations/Go_To_Considered_Harmful.html

- Declarative = emphasizes on what is to be computed and not on how.
- Declarative = expressing a command, commanding

It might sound abstract. How can an algorithm focus more on the ‘what’ than on the ‘how’? To make things clear we’ll be comparing an imperative solution with a functional solution below.

Imperative solution

Below you find the classical imperative solution for the problem. It mixes some patterns that many people have acquired while learning to program.

It iterates over all the stores, and if the store is within the maximum distance it goes into a block of code with two functions.First we find the business rule that describes our discount.

Next we recognize a pattern to calculate a minimum value. The pattern exist out of two parts: (1) On top `min` is defined as a utility variable to hold the last known minimum, (2) while within the loop min is updated with a new value if that new value is lower than the current known value.

```
def find_cheapest_nearby (stores)
  min = 0
  stores.each do |store|
    if store[:d] < 5.0 then
      store[:price] = store[:price] * 0.9 if store[:name] == "shell"
      min = store[:price] if store[:price] < min
    end
  end
  min
end
```

Was that hard to analyze? Not really. But that’s mainly because we already new what the algorithm is expected to do. If the title of the method would be less clear and you would be looking at this code snippet without a lot of context. It would already be a bit harder. But still, as the algorithm is short and simple, a trained developer would be able to recognize the patterns and derive from that it’s function. Worst case we would resort to playing the computer ourselves in our head, iterating over the different instructions.

In the mean time, many of the readers might have been feeling uncomfortable. If you are a bit sharp while reading this, you’ve probably noted at least one ‘bug’ in the above code.

First of A-bombll, the pattern for calculating the minimum value wasn’t applied correctly. As `min` was initialized as `0`, unless there are negative prices, the outcome of the previous algorithm will always be `0`.

Second, and this one is much more scary, the routine will function correctly the first time it is run, however on repeated execution it will start returning wrong results. Why? Because the algorithm changes the list prices on each run.

Fixes are easy. So let’s get rid of those mistakes. We’ll initialize `min` with the first value and introduce another variable to hold our price.

```
def best_price_nearby (stores)
  min = stores[0][:price]
  stores.each do |store|
    if store[:d] < 5.0 then
      price = store[:price]
      price = price * 0.9 if store[:name] == "shell"
      min = price if price < min
    end
  end
  min
end
```

Functional Solution

Let's take a look now at the more functional solution variant. First of all, don't get frightened if you see some constructs which look new to you. They will get clear soon enough. And chances are that your best guess on what they stand for, will just be correct.

```
def best_price_nearby (stores)

  near = ->(x) { x[:d] < 5.0 }
  myPrice = ->(x) { x[:name] == "shell" ? x[:price]*0.9 : x[:price] }

  stores.
    find_all(&near).
    collect(&myPrice).
    min
end
```

Let's break it down step by step. The first expression we meet is a definition of a function that decides if the shop is within our reach.

```
near = ->(d, x) { x[:d] < 5.0 }
```

This is a so called *Predicate* function, a function which returns `true` or `false`.

Next we find an expression that defines a function that given a shop returns our price.

```
  myPrice = ->(x) { x[:name] == "shell" ? x[:price]*0.9 : x[:price] }
```

And now let's break down the final expression. This is where we see list processing in action for the first time. The first step is the following.

```
  stores.find_all(&near)
```

What does it do? Well, we all expect it to return the stores that are near. How does it do that? Later we’ll be looking at find_all in more detail, but find_all is a method that given a Predicate function - a function that returns `true` or `false` - returns all elements of the list for which the Predicate function returns `true`.

But than it goes on. For all the shops that are near, it collects my price.

```
  stores.find_all(&near).collect(&myPrice)
```

How does it do that? Later we’ll look in a lot more detail to the `collect` function. In short it converts our list of nearby shops into a list of prices by applying the `myPrice` function to each of the elements.

We are almost there. To get the best price from the list of prices, we simply use the `min` function. A standard function that is available on all lists that contain comparable elements.

```
  stores.find_all(&near).collect(&myPrice).min
```

Conclusions

Let’s repeat the functional variant, to look at it again. What do we see?

```
def best_price_nearby (stores)

  near = ->(x) { x[:d] < 5.0 }
  myPrice = ->(x) { x[:name] == "shell" ? x[:price]*0.9 : x[:price] }

  stores.
    find_all(&near).
    collect(&myPrice).
    min
end
```

First of all, there are actually some things we don’t see.

- We don’t see any more a classical for loop.
- We don’t see utility variables.
- We don’t see a pattern to calculate the minimum value
- We don’t see a lot of code that we would need to ‘interprete’ to know what’s going on.

We also note the following

- the definition of what it means for a shop to be near is nicely separated from the rest of the code.
- the *business rule* that calculates my price from the list price is also nicely separated from the code
- the ‘program’ that finds all near shops and selects from those the minimum price can almost be read as natural English.

Declarative programming is counterintuitive when you’re used to imperative programming, at least when you are asked to write down an algorithm. This all goes back to the very early days of computer science, where scientist where giving the machine sequences of *instructions*. Because deep down, computers still are dumb, and just are very fast in executing basic instructions, we tend to think and train people that software is about chaining basic instructions into programs.

But from a reader and maintenance point of view, functional coding makes code more readable and hence less error prone and easier to maintain. Towards the compiler the software should express correctly and formally its precise function. But remember that the code you write will be read many many times over the lifetime of the software by yourself or human colleagues. The real quality of software lies in the expressiveness of your code on it’s goal towards Humans, not compilers. Because that in the end will largely influence strongly the development and maintenance cost of your software.

Putting functional programming in context. Explain that functional programming is only one way to manage complexity.

Software and Complexity and it’s cost

At some point in a programming curriculum, one gets introduced to complexity, typically the algorithmic or computational complexity. In **computational** complexity theory, the amounts of resources required for the execution of algorithms is studied. The most popular types of computational complexity are the time complexity and the space complexity of a problem. Computational complexity is an important concept to well understand, and we will be talking about computation complexity a lot more in the last part where we will be talking about performance. What is often not addressed, is that there are other possibly even more relevant forms of complexity in the domain of software engineering.

System complexity is a measure of the interactions of the various elements of the software. This differs from the computational complexity in that it is a measure of the design of the software.

```
System complexity = Number of possible interactions in a system of n components
= n!
```

Many things can put a project off course, bureaucracy, unclear objectives, lack of resources, to name a few, but it is the approach to design that largely determines how complex software can become. When complexity gets out of hand, the software can no longer be understood well enough to be easily changed or extended. By contrast, a good design can make opportunities out of those complex features. Associated with, and dependent on the complexity of an existing

> program, is the complexity associated with changing the program.
>
> **Domain Complexity**. Some of the design factors are technological, and a great deal of effort
>
> has gone into the design of networks, databases, and other technical dimension of software. Books have been written about how to solve these problems. Developers have cultivated their skills. Yet the most significant complexity of many applications is not technical. It is in the domain itself, the activity or business of the user. When this domain complexity is not dealt with in the design, it won’t matter that the infrastructural technology is well-conceived. A successful design must systematically deal with this central aspect of the software.
>
> **Accidental Complexity** is a very interesting one. It’s complexity that architects and developers create themselves. It’s that complexity which arises in computer programs or in the development process which is not essential to the problem being solved. Accidental complexity is caused by a suboptimal approach to the problem’s resolution.

Software engineering as a discipline is largely is about techniques to manage different kinds of complexity, through techniques and processes. If we forget about the processes for a moment - things like agile, design patterns and BDD - a modern developer has a huge set of tools available when designing software:

- Service Orientation
- Layering
- Modules/Libraries
- Functions/Procedures
- Domain Specific Languages
- Object Orientation
- Frameworks
- Aspect Orientation
- Domain Driven Design

We will zoom in to only a selection of the above techniques. But as a software professional you should try to master all of those techniques, they are all equally important, depending on the problem you are faced with. Always try to keep in mind, what goal they should be serving. As long as when applied your system complexity is reduced, they are likely applied rightly. If you ever see any of the techniques applied in such a way that system complexity is increased (number of possible interactions within the system), than probably it means that, whatever the technique is, it’s being wrongly applied in that specific context.

Short History of Functional Programming

- Lambda calculus: 1930’s (Alonzo Church)

- Lisp: 1950's, multi-paradigm language strongly inspired by lambda-calculus (symbolic manipulation, rewriting)
- ML-family: 1970's, general purpose functional programming language with type inference
- Haskell: 1987, 1998, 2010, open standard for functional programming research

The world of functional programming languages

Clean, F#, Scheme, Scala, Clojure, Erlang, …

SQL

SQL is not imperative because the process of HOW queries and relationships are resolved are not defined by the programmer, but rather by the compiler/optimizer/interpreter. SQL is a declarative language - In SQL, you declare relationships. This builds up a data structure (that again is not physically defined with the language but by its implementation) using inserts, updates, and deletes.

Use of the relations is then done using queries (SELECT statements), which are functional in that they do not have side effects.

Functional programming centers around a few core ideas:

- functions are first-class citizens (that is, they can be used as values, as inputs to other functions, and as output from other functions)
- higher-order functions (functions that operate on functions, or functions that return functions)
- purity (a pure function is one that has no side effects; a pure function cannot do any I/O, it cannot read nor modify any global state, and it cannot take non-const reference arguments. Pure functions are especially interesting because they will always produce the same output given the same inputs)

SQL certainly doesn't revolve around functions as the main tool for modeling things, but it does somewhat embrace the purity idea - the same query run on the same database will yield the same result, every time (except for ordering). But calling SQL a 'functional' language is a bit of a stretch though.

XSLT

The XSLT language is declarative: rather than listing an imperative sequence of actions to perform in a stateful environment, template rules only define how to handle a node matching a particular XPath-like pattern, if the processor should happen to encounter one, and the contents of the templates effectively comprise functional expressions that directly represent their evaluated form: the result tree, which is the basis of the processor's output.

Although XSLT is based on functional programming ideas, it is not a full functional programming language, it lacks the ability to treat functions as a first class data type. It has elements like lazy evaluation to reduce unneeded evaluation and also the absence of explicit loops.