The Art of Functional Programming

with examples in OCaml, Haskell, and Java

Minh Quang Tran, PhD

The Art of Functional Programming

Copyright © 2024 Minh Quang Tran
All rights reserved.

About the author:

Minh Quang Tran has over 20 years of experience studying software development and working in the software industry. He has worked in various tech startups and large software companies in Europe. He holds a Bachelor of Computer Science from Furtwangen University, Germany, an M.Sc. in Computer Science from McMaster University, Canada, and a Ph.D. in Computer Science from the Technical University of Berlin, Germany. His interest lies in understanding and mastering the fundamentals and principles that transcend programming languages, frameworks, and tools.

1 Introduction

1.1 About This Book

1.1.1 Book description

Welcome to *The Art of Functional Programming* book!

Functional programming is a powerful and elegant programming paradigm. Initially only popular among university researchers, it's gained much traction in the software industry in the last few years. From big companies to start-ups, engineers and managers have realized that functional programming excels at abstraction and composition. Functional programming allows for highly concise solutions with increased safety. This has led to rising demand for software engineers with functional programming skills. This book will help you move your programming skills to the next level.

This book is grounded on my beliefs about software development resulting from my reflection after many years of studying and working in the software industry. First, there are tons of programming languages, frameworks, and tools out there – with many more coming in the future. The only way to stay ahead of the game in this vast and quickly changing software industry is to master the fundamentals and principles that cut across programming languages, frameworks, and tools. In the case of functional programming, learning to adopt the functional way of solving problems is much more productive than memorizing how to write functional code in a particular language. This book teaches this functional way of thinking.

We'll also learn many fundamental techniques from programming languages, such as parsing, compilation, and type checking.

Second, I believe that a technique or tool is not very useful if it does not help us solve real-world problems and make our lives easier. In this book, we'll only look at examples and exercises that are typically encountered in a programmer's day-to-day job. Furthermore, an entire chapter is dedicated to applying what we've learned to real-world scenarios. In particular, we'll use functional programming to process collections of data for an e-commerce application and handle the JSON datatype.

I like to think of the duality of striving to grasp the fundamental principles of the functional paradigm while applying it to real-world problems pragmatically as the yin and yang of the journey to master functional programming.

Figure 1: Yin and yang as a symbol of duality

Throughout this book, we'll mainly utilize OCaml, supplemented occasionally by Haskell, to illustrate the key concepts and techniques inherent to functional programming throughout the book. To highlight the differences between functional and imperative paradigms, we'll draw comparisons with Java.

OCaml and Haskell, both members of the ML family, are uniquely suited to demonstrating the power and elegance of the functional programming paradigm, having been designed with functional programming at their core. Peter Norvig, in his insightful essay *Teach Yourself Programming in Ten Years*, advises burgeoning software engineers to "learn at least a half dozen programming languages. Include … one [language] that emphasizes functional abstraction (like Lisp or ML or Haskell)."

Now, you may be wondering, "I'm a Go, Java, JavaScript, Kotlin, Swift, Python, Scala, X programmer. Can I still benefit from this book?" I wholeheartedly assure you that the answer is a resounding "Yes!"

I've titled this book "The Art of Functional Programming" precisely because it teaches the timeless principles of functional programming that transcend specific programming languages. We'll delve into the essence of programming, exploring concepts like abstraction, composition, code reusability, and generic programming, etc., and demonstrate how functional programming enables us to embody these principles elegantly. You can apply them in any programming language that supports the functional programming style.

Here is the summary of the chapters in this book:

- In Chapter 1: Introduction, we'll start the book with an introduction to functional programming. In particular, we'll see how it can overcome some of the inherent weaknesses of the imperative programming paradigm. We'll also discuss why functional programming matters to any software engineer.

- In Chapter 2: Expressions – The Building Blocks of Functional Programs, we examine expressions and how to build complex expressions from simpler ones. Three aspects of expressions – syntax, types, and semantics – will be covered. Along the way, we'll gain a much deeper understanding of how programming languages work, including parsing, type checking, interpretation, and compilation.
- In Chapter 3: Building Abstractions with Functions, we'll get to know lambda calculus – a mathematical model serving as the foundation of all functional programming languages. We'll learn how to capture computation patterns as functions. Finally, we'll discuss various techniques for working with functions such as currying, recursion, and higher-order functions.
- In Chapter 4: Compound Data Types, we'll focus on the compound data types typically found in functional programming languages, such as tuples and lists. Furthermore, we'll use algebraic data types to represent hierarchical data, and pattern matching to extract data from compound data types.
- In Chapter 5: Common Computation Patterns, we'll dive into some of the most common computation patterns, such as map, filter, fold, and zip. These functions capture highly general computation patterns on lists and other data structures that can be reused to formulate many other functions.
- In Chapter 6: Dataflow Programming with Functions, we'll go

over dataflow programming, a programming paradigm that emphasizes composing programs from existing components. We'll learn how functional programming allows us to do dataflow programming elegantly and reap all its benefits.

- In Chapter 7: Applying Functional Programming in Practice, we'll apply what we've learned to process collections of data commonly found in mobile and web applications, as well as backend services. Furthermore, we'll use functional programming to represent and handle JSON.
- In Chapter 8: Conclusion, we'll conclude the book and review what we've learned and what are the next steps you can pursue after reading this book.

1.1.2 Intended audience

This is a beginner and intermediate book aimed at software engineers, engineering managers, or computer science students interested in understanding the essence of functional programming. It is also an excellent fit for individuals who are currently preparing for coding interviews and want to improve their problem-solving skills.

1.2 A Bite of Functional Programming

Functional programming is a **programming paradigm** — a style or a way of thinking when writing software programs. There are many programming paradigms available, some of which are presented in the following diagram.

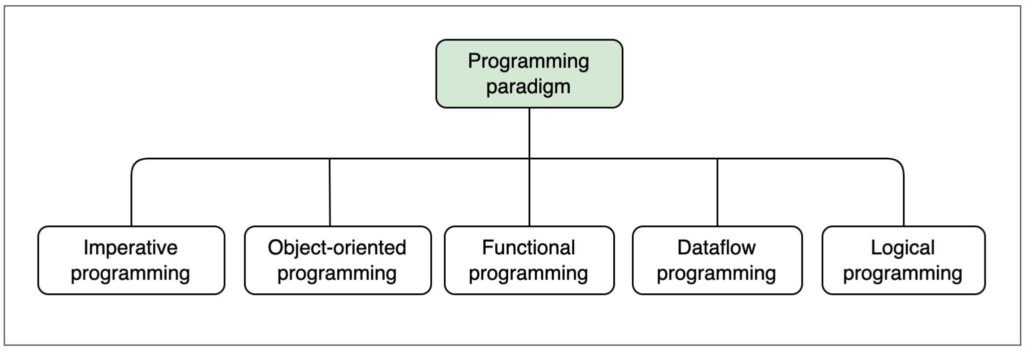

Figure 2: Programming paradigms

The **imperative programming paradigm** is the oldest, and still the most popular, paradigm used today. As the term "imperative" indicates, this paradigm models a program as a sequence of commands that change a program's state – "first do this, then do that." Even if we follow the object-oriented programming paradigm by encapsulating logic into classes, we'll likely implement the class methods in the imperative style.

Why has imperative programming dominated the programming world? In his Turing Award lecture titled *Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs*, the computer scientist John Backus gives an insightful answer. According to his observation, the reason can be traced back to the **von Neumann architecture**. Named after the mathematician John von Neumann, who took inspiration from the Turing machine, it is a computer architecture used by most computers produced today.

In the following, we'll review the von Neumann architecture and discuss several problems of the imperative programming paradigm due to its tight coupling with this architecture. Then we'll see how the functional programming paradigm can elegantly overcome those problems.

1.2.1 Von Neumann architecture

In its simplest form, a von Neumann computer consists of a Central Processing Unit (CPU), a memory, and a bus that connects them. The CPU acts as the brain of the computer with the ability to execute a predefined set of machine code instructions. These instructions are in binary form, consisting of 0s and 1s, and do a very primitive thing, such as adding two numbers or testing whether a number equals zero or not. The CPU has a handful of registers to store data needed when executing an instruction.

The memory is the place where a program and its data are stored. A program is a sequence of machine code instructions. The CPU and the memory are connected via a bus. Due to this, a subset of machine code instructions is dedicated to loading data from memory onto the registers or storing the data from the register onto the memory.

The following diagram illustrates the von Neumann architecture.

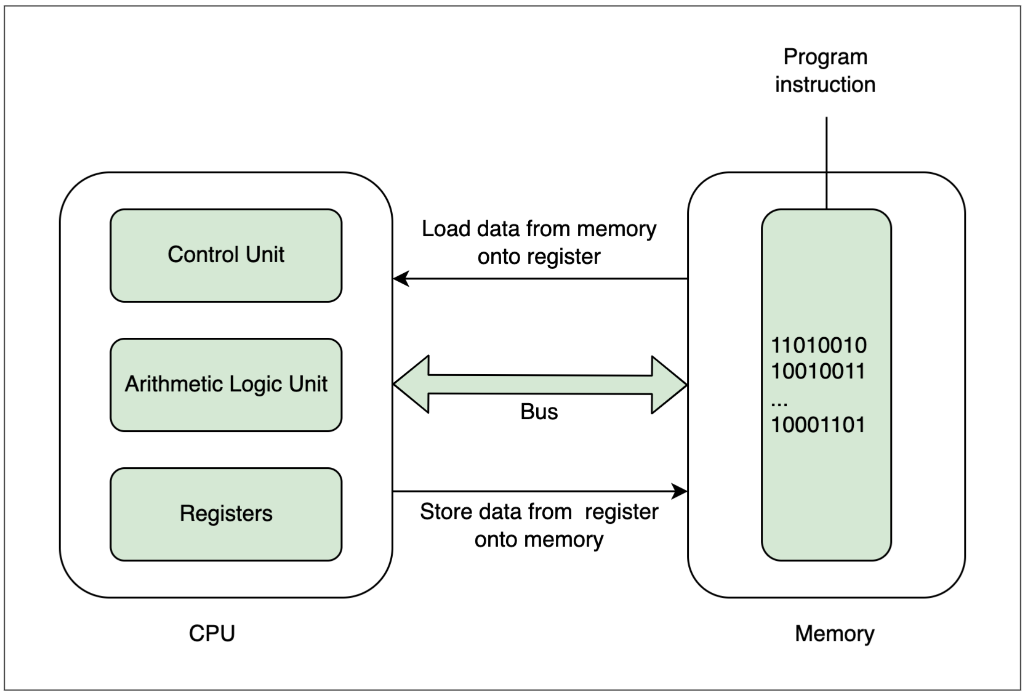

Figure 3: The von Neumann architecture

So, how does a program run? It's quite simple. The CPU runs the program following a mechanical **fetch-execute cycle**. First, it fetches the first instruction in the program and executes it. Then it fetches the next instruction and executes it, and this cycle continues on. Some instructions affect the order of execution. For instance, a jump instruction directs the CPU to jump back to a previous point of the instruction sequence. A branch instruction tells the CPU to branch to a particular instruction if some condition is true, for example, if two registers have the same values. These instructions are typically combined to implement loops and if-statements.

1.2.2 Low-level nature of imperative programming

Let’s stop for a moment and reflect on the thought process when programming for the von Neumann computer. A program is constructed as a sequence of instructions whose main task is to move data back and forth between the CPU and memory. These instructions also perform arithmetic and logical operations. The primary concern is how to update the memory cells in a stepwise manner. This model of programming for the von Neumann architecture is the essence of what imperative programming is about.

Consider, for instance, an example of calculating the sum of squares of the first `n` numbers in the imperative style.

```
int sum = 0; i = 0;
while (i < n) {
    i = i + 1;
    sum = sum + i * i;
}
```

This program might be written with a high-level language, such as C, Java, or Python. Yet, the code is nothing more than a sequence of statements telling the physical computer how to update the memory. In particular, the variables `sum` and `i` correspond to the memory cells. An assignment statement, such as `i = i + 1`, equals moving the data from the memory to the CPU’s registers, asking the CPU to perform the addition, and moving the data from the registers to the memory to update the memory cell occupied by `i`. The **`while`** loop corresponds to how a physical computer uses branch and jump instructions to execute instructions repeatedly so long as the condition is still valid.

Its coupling with the von Neumann architecture makes the imperative programming paradigm quite limited when it comes to forming abstractions and compositions when constructing programs.

1.2.3 Functional programming can do better

Let's use the functional programming paradigm to calculate the sum of squares of the first `n` numbers and compare the functional version with the imperative one.

Imperative style

```
int sum = 0; i = 0;
while (i <= n) {
   i = i + 1;
   sum = sum + i * i;
}
```

Functional style

```
(fold (+) 0 . map square) [1..n]
```

The imperative program is just a sequence of low-level statements for updating variables rather than constructed from simpler parts. The loop is a single unit and cannot be broken into smaller components. In contrast, the functional program is built from reusable parts. Only the functions – `square`, addition `(+)`, and the initial value `0` – are specific to this program. The rest is assembled from general-purpose components, such as `map`, `fold`, and function composition. In particular, `map` applies a given function to all list elements, whereas `fold` combines elements in a list using a function, starting from an initial value. The function composition operator, `.`, allows us to turn the output of one function into the input of another one.

In fact, we can view the functional program above as a dataflow program that emphasizes the composability of this solution.

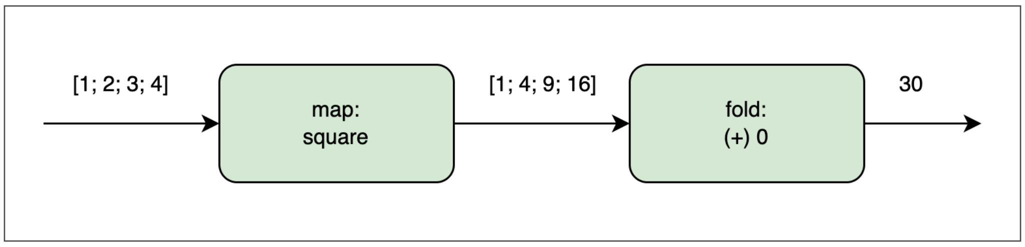

Figure 4: Functional program viewed as dataflow program

Another critical difference is that the first program is imperative, while the second one is descriptive. More specifically, the imperative program contains a sequence of commands stating how to initialize variables and update them in each step. The downside is we must mentally execute it to understand what it does, which requires a higher cognitive load. The functional program is declarative because it describes what the program does rather than how each step is computed. We can grasp the program based on its structure in one fell swoop without mentally executing it.

Now assume we would like to write another program that computes the sum of squares of only prime numbers between `1` and `n`. Using the imperative style, we can copy the code of the old program and add an `if` statement.

```
int sum = 0; i = 0;
```

```
while (i <= n) {
    i = i + 1;
    if (isPrime(i)) {
        sum = sum + i * i;
    }
}
```

Here, we assume `isPrime` is a method that returns **`true`** if the argument is a prime and returns **`false`** otherwise.

However, in the functional programming paradigm, the solution is much more elegant.

```
(fold (+) 0 . map square . filter isPrime)
    [1..n];;
```

Compared to the imperative version, this program has a higher degree of **composability**, with functions readily combined into more powerful constructs. Here, we plug in another general-purpose function called `filter` to choose prime numbers from the list before passing them to `map` and then `fold`.

Let's look at the data flow diagram below:

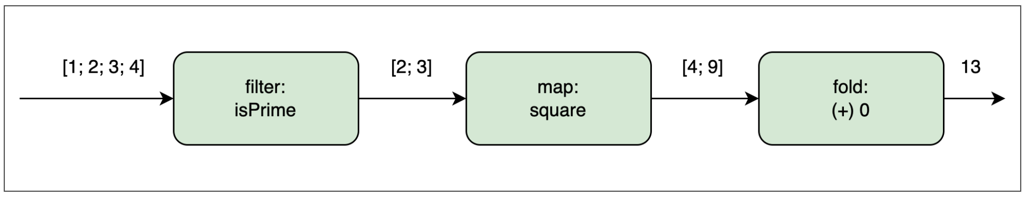

Figure 5: Calculate sum of squares of prime numbers in functional style

It is important to emphasize that it's by no means the purpose of this comparison to show that functional programming is *better* than imperative programming. As with any tool, way of thinking, or method of solving problems, each programming paradigm is more suitable for particular situations and less for others. Developing an intuition to decide when to use what tool and method is part of becoming an expert in any field, including programming. This book aims to help us develop this intuition so that we'll know when to utilize the power of functional programming and when not to.

1.3 Why Functional Programming Matters?

Are you a professional software engineer or aspire to become one? If yes, functional programming is undoubtedly among the most valuable skills to learn. This section will show why every software engineer should learn functional programming and why the best time to start is today.

1.3.1 Powerful problem-solving tool

Learning functional programming means you acquire a new tool to solve problems. As with any craftsmanship, the more tools you have mastered and have at your disposal, the more skillful you become. In his insightful essay *Teach Yourself Programming in Ten Years*, Peter Norvig recommends aspiring software engineers to "learn at least a half dozen programming languages. Include … one [language] that emphasizes functional abstraction (like Lisp or ML or Haskell)". Both programming languages used in this book, OCaml and Haskell, are ML languages that emphasize functional abstraction.

Functional programming is a great tool to master because it can elegantly solve many programming problems in various domains. For instance, functional programming excels at applications that deal with hierarchical structures such as JSON and XML. Functional programming is also well suited to data processing in mobile apps, web apps, or backend services, especially when filtering, transforming, and aggregating data.

Even if we don't use functional programming in our day-to-day work, we still hugely benefit from learning it. Functional programming focuses on composition, or building complex programs from simpler ones, as well as on abstraction, or defining highly reusable general functions capturing common computation patterns. These are vital techniques for managing complexity when structuring code and building large software systems. As a result, learning the functional programming paradigm sharpens our ability to design software and write clean reusable code.

1.3.2 The trend from imperative towards the declarative paradigm

The software industry has witnessed a gradual shift towards functional programming in recent years. Non-functional mainstream programming languages, such as Java, keep introducing features to write functional code. New programming languages, such as Elm, Elixir, Scala, Swift, and Kotlin, support functional programming from the ground up. Furthermore, more and more frameworks and libraries are heavily based on the functional programming paradigm, such as ReactiveX and Akka Streams. This implies that there is an increasing demand for software engineers with functional programming skills.

Interestingly, the trend towards functional programming is just a part of an overall transition from the imperative to the declarative paradigm in the software industry. Some of them are declarative UI, declarative build systems, declarative build pipelines, and even declarative deployment infrastructure.

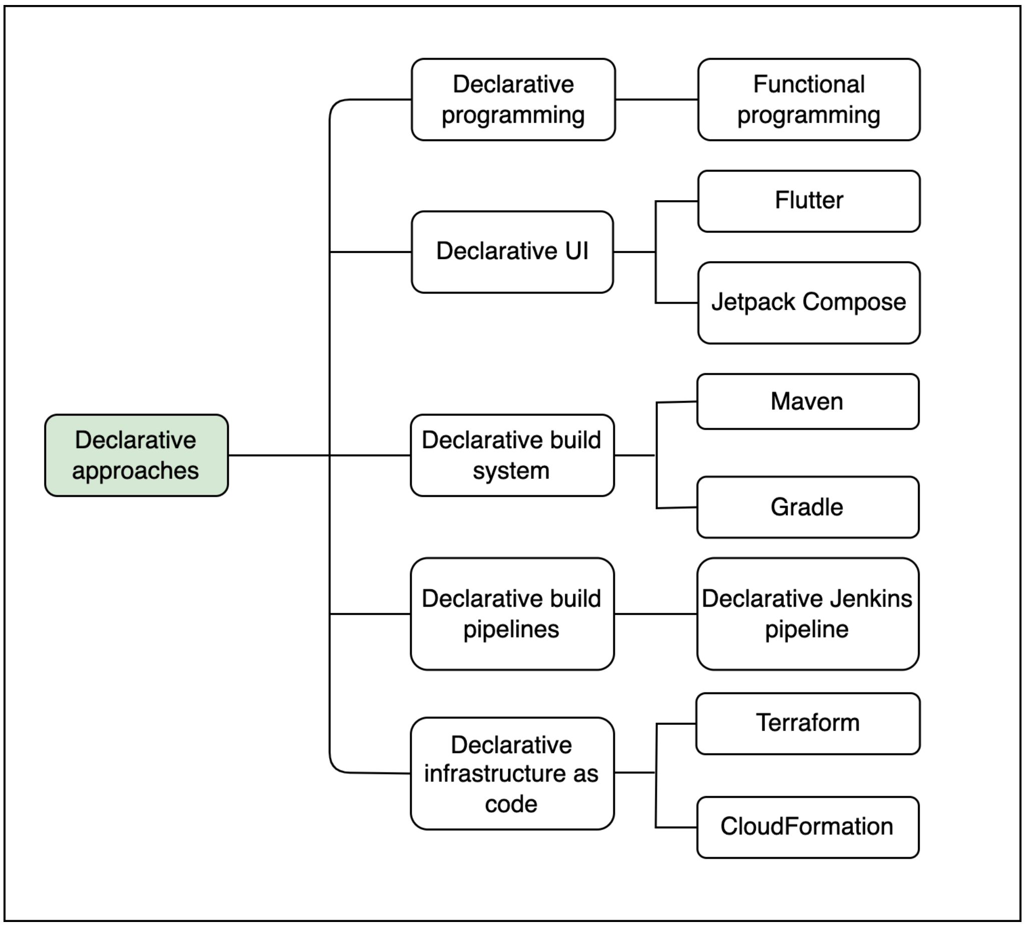

Figure 6: Use cases for declarative paradigm in software industry

Take build systems, for instance. The two most popular build systems today, Maven and Gradle, follow the declarative approach. We tell the system what we want to achieve, and the build system will figure out how to actually do it. This contrasts with imperative build systems like Ant, where we explicitly specify how the system should perform the build by listing an ordered sequence of the statements or commands.

The evolution from imperative to declarative paradigms is also a notable shift in front-end development. Many of you probably still remember not so long ago, web development was primarily driven by jQuery, a JavaScript library that facilitated direct DOM (Document Object Model) manipulation. This approach was inherently imperative, as we explicitly stated how and when to modify a DOM element.

However, this imperative approach has been largely superseded by a more intuitive, declarative paradigm championed by modern UI libraries such as React and Vue. Instead of dictating each specific change to the DOM, we now merely declare how a page should look. It's then the responsibility of these libraries to figure out the optional sequence of DOM manipulations to achieve the desired outcome.

Declarative systems, whether for describing build systems or crafting UI components, offer numerous advantages over their imperative counterparts. In particular, they allow us to define elements at a high level without getting entangled in the intricate details of implementation. Moreover, they make it easier to create complex components from smaller ones. The same can be said about declar-