

Chapter 2

Introduction

Programming is a creative process in which exciting problems may be solved and new tools and applications may be created. With programming skills you can create high-level applications to run on a mobile device that interacts with other users, databases, and artificial intelligences; you may create programs that run on super computers for simulating weather systems on alien planets or social phenomenons in the internet economy; and you may create programs that run on small custom-made hardware for controlling your home appliances.

2.1 How to learn to program

In order to learn how to program there are a couple of steps that are useful to follow:

1. Choose a programming language: It is possible to program without a concrete language, but your ideas and thoughts must be expressed in some fairly rigorous way. Actually, theoretical computer science typically does not rely on computers nor programming languages, but uses mathematics to prove properties of algorithms. However, most computer scientists program, and with a real language, you have the added benefit of checking your algorithm and hence your thoughts rigorously on a real computer. This book teaches a subset of F#. The purpose is not to be a reference guide to this language, but to use it as a vessel to teach you, the reader, how to convert your ideas into programs.
2. Learn the language: A computer language is a structure for thought, and it influences which thoughts you choose to implement as a program, and how you choose to do it. Any conversion requires you to acquire a sufficient level of fluency, for you to be able to make programs. You do not need to be a master in F# nor to know every corner of the language, and you will expand your knowledge as you expose yourself to solving problems in the language, but you must invest an initial amount of time and energy in order to learn the basics of the language. This book aims at getting you started quickly, which is why we intentionally are teaching a small subset of F#. On the net and through other works, you will be able to learn much more.
3. Practice: If you want to be a good programmer, then there is only one way: practice, practice, practice! It has been estimated that to master anything, then you have to have spent at least 10000 hours of practice, so get started logging hours! It of course matters, what you practice. This book teaches 3 different programming themes. The point is that programming is thinking, and the scaffold that you use, shapes your thoughts. It is therefore important to recognize this scaffold, and to have the ability to choose that which suits your ideas and your goals best. And the best way to expand your abilities is to both sharpen your present abilities, push yourself into

new territory, and trying something new. Do not be afraid to make errors or be frustrated at first. These are the experiences that make you grow.

4. Solve real problems: I have found that using my programming skills in real situations with customers demanding solutions, that work for them, has allowed me to put into perspective the programming tools and techniques that I use. Often customers want solutions that work, are secure, are cheap, and delivered fast, which has pulled me as a programmer in the direction of “if it works, then sell it”, while on the longer perspective customers also want bug fixes, upgrades, and new features, which require carefully designed code, well written test-suites, and good documentation. And as always, the right solution is somewhere in between. Regardless, real problems create real programmers.

2.2 How to solve problems

Programming is the act of solving a problem by writing a program to be executed on a computer. A general method for solving problems was given by George Pólya [9] and adapted to programming is:

Understand the problem: To solve any problem it is crucial that the problem formulation is understood, and questions like: What is to be solved? Do you understand everything in the description of the problem. Is all information for finding the solution available or is something missing?

Design a plan: Good designs mean that programs are faster to program easier to debug and maintain. So before you start typing a program consider things like: What are the requirements and constraints for the program? Which components should the program have? How are these components to work together? Designing often involves drawing a diagram of the program, and writing pseudo-code on paper.

Implement the plan: Implementation is the act of transforming a program design into code. A crucial part of any implementation is choosing which programming language to use. Also, the solution to many problems will have a number of implementations which vary in how much code they require, to which degree they rely on external libraries, which programming style they are best suited for, what machine resources they require, and what their running times are. With a good design, then the coding is usually easy, since the design will have uncovered the major issues and found solutions for these, but sometimes implementation reveals new problems, which requires rethinking the design. Most implementations also include writing documentation of the code.

Reflect on the result: A crucial part in any programming task is ensuring that the program solves the problem sufficiently. E.g., what are the program's bugs, is the documentation of the code sufficient and relevant for its intended use. Is the code easily maintainable and extendable by other programmers. Are there any general lessons to be learned from or general code developed by the programming experience, which may be used for future programming sessions?

Programming is a very complicated process, and Pólya's list is a useful guide, but not a failsafe approach. Always approach problem solving with an open mind.

2.3 Approaches to programming

This book focuses on 3 fundamentally different approaches to programming:

Imperative programming, which is a type of programming where *statements* are used to change the

- Imperative programming
- statements

program's *state*. Imperative programming emphasises *how a program shall accomplish a solution* and less on *what the solution is*. A cooking recipe is an example of the spirit of imperative programming. Almost all computer hardware is designed to execute low-level programs written in imperative style. The first major language was FORTRAN [6] which emphasized imperative style of programming.

- state
- Declarative programming
- Functional programming
- functional programming
- functions
- expressions
- Structured programming
- Object-orientered programming
- objects

Declarative programming, which emphasises *what a program shall accomplish* but not *how*. We will consider Functional programming as an example of declarative programming language. A *functional programming* language evaluates *functions* and avoids state changes. The program consists of *expressions* instead of statements. As a consequence, the output of functions only depends on its arguments. Functional programming has its roots in lambda calculus [2], and the first language emphasizing functional programming was Lisp [7].

Structured programming, which emphasises organisation of code in units with well defined interfaces and isolation of internal states and code from other parts of the program. We will focus on Object-oriented programming as the example of structured programming. *Object-orientered programming* is a type of programming, where the states and programs are structured into *objects*. A typical object-oriented design takes a problem formulation and identifies key nouns as potential objects and verbs as potential actions to be taken on objects. The first object-oriented programming language was Simula 67 developed by Dahl and Nygaard at the Norwegian Computing Center in Oslo.

Most programs follows a single programming paradigm as, e.g., one of the above, but are a mix. Nevertheless, this book will treat each paradigm separately to emphasize their advantages and disadvantages.

2.4 Why use F#

This book uses F# also known as Fsharp, which is a functional first programming language that also supports imperative and object-oriented programming. It was originally developed for Microsoft's .Net platform, but is available as open source for many operating systems through Mono. As an introduction to programming, F# is a young programming language still under development, with syntax that at times is a bit complex, but it offers a number of advantages:

Interactive and compile mode F# has an interactive and a compile mode of operation: In interactive mode you can write code that is executed immediately in a manner similarly to working with a calculator, while in compile mode, you combine many lines of code possibly in many files into a single application, which is easier to distribute to non F# experts and is faster to execute.

Indentation for scope F# uses indentation to indicate scope: Some lines of code belong together, e.g., should be executed in a certain order and may share data, and indentation helps in specifying this relationship.

Strongly typed F# is strongly typed, reducing the number of runtime errors: F# is picky, and will not allow the programmer to mix up types such as integers and strings. This is a great advantage for large programs.

Multi-platform F# is available on Linux, Mac OS X, Android, iOS, Windows, GPUs, and browsers via the Mono platform.

Free to use and open source F# is supported by the Fsharp foundation (<http://fsharp.org>) and sponsored by Microsoft.

Assemblies F# is designed to be able to easily communicate with other .Net and Mono programs through the language-independent, platform-independent bytecode called Common Intermediate

Language (CIL) organised as assemblies. Thus, if you find that certain parts of a program are easy to express in F# and others in C++, then you will be able to combine these parts later into a single program.

Modern computing F# supports all aspects of modern computing including Graphical User Interfaces, Web programming, Information rich programming, Parallel algorithms, ...

Integrated development environments (IDE) F# is supported by major IDEs such as Visual Studio (<https://www.visualstudio.com>) and Xamarin Studio (<https://www.xamarin.com>).

2.5 How to read this book

Learning to program requires mastering a programming language, however most programming languages contains details that are rarely used or used in contexts far from a specific programming topic. Hence, this book takes the approach to start with an introduction to the most basic concepts of F# in ??, followed by the 3 programming paradigms in ??–?? while gradually expanding the introduction of F# syntax and semantics. In ?? are a number of general topics given for reference. The disadvantage of this approach is that no single part contains a reference guide to F# and F# topics are revisited and expanded across the book. For further reading please consult <http://fsharp.org>.