Learning to program with F#

Jon Sporring

Department of Computer Science,
University of Copenhagen

January 22, 2018
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.2</td>
<td>Function bindings</td>
<td>44</td>
</tr>
<tr>
<td>6.3</td>
<td>Operators</td>
<td>50</td>
</tr>
<tr>
<td>6.4</td>
<td>Do bindings</td>
<td>51</td>
</tr>
<tr>
<td>6.5</td>
<td>The printf function</td>
<td>52</td>
</tr>
<tr>
<td>6.6</td>
<td>Reading from the console</td>
<td>55</td>
</tr>
<tr>
<td>6.7</td>
<td>Variables</td>
<td>55</td>
</tr>
<tr>
<td>6.8</td>
<td>Reference cells</td>
<td>58</td>
</tr>
<tr>
<td>6.9</td>
<td>Tuples</td>
<td>61</td>
</tr>
<tr>
<td>7</td>
<td>In-code documentation</td>
<td>65</td>
</tr>
<tr>
<td>8</td>
<td>Controlling program flow</td>
<td>71</td>
</tr>
<tr>
<td>8.1</td>
<td>While and for loops</td>
<td>71</td>
</tr>
<tr>
<td>8.2</td>
<td>Conditional expressions</td>
<td>76</td>
</tr>
<tr>
<td>8.3</td>
<td>Programming intermezzo: Automatic conversion of decimal to binary numbers</td>
<td>78</td>
</tr>
<tr>
<td>9</td>
<td>Organising code in libraries and application programs</td>
<td>81</td>
</tr>
<tr>
<td>9.1</td>
<td>Modules</td>
<td>81</td>
</tr>
<tr>
<td>9.2</td>
<td>Namespaces</td>
<td>84</td>
</tr>
<tr>
<td>9.3</td>
<td>Compiled libraries</td>
<td>86</td>
</tr>
<tr>
<td>10</td>
<td>Testing programs</td>
<td>90</td>
</tr>
<tr>
<td>10.1</td>
<td>White-box testing</td>
<td>92</td>
</tr>
<tr>
<td>10.2</td>
<td>Black-box testing</td>
<td>95</td>
</tr>
<tr>
<td>10.3</td>
<td>Debugging by tracing</td>
<td>98</td>
</tr>
<tr>
<td>11</td>
<td>Collections of data</td>
<td>107</td>
</tr>
<tr>
<td>11.1</td>
<td>Strings</td>
<td>107</td>
</tr>
<tr>
<td>11.1.1</td>
<td>String properties</td>
<td>107</td>
</tr>
<tr>
<td>11.1.2</td>
<td>String module</td>
<td>108</td>
</tr>
<tr>
<td>11.2</td>
<td>Lists</td>
<td>110</td>
</tr>
<tr>
<td>11.2.1</td>
<td>List properties</td>
<td>113</td>
</tr>
<tr>
<td>11.2.2</td>
<td>List module</td>
<td>113</td>
</tr>
</tbody>
</table>
## CONTENTS

### 11.3 Arrays
- Array properties and methods .................................................... 120
- Array module ........................................................................... 120

### 11.4 Multidimensional arrays
- Array2D module .................................................................. 128

### 12 The imperative programming paradigm .................................. 130
- Imperative design ................................................................... 131

### 13 Recursion .......................................................................... 132
- Recursive functions ................................................................ 132
- The call stack and tail recursion .......................................... 133
- Mutual recursive functions ..................................................... 138

### 14 Programming with types ................................................... 140
- Type abbreviations ................................................................. 140
- Enumerations ........................................................................ 141
- Discriminated Unions ............................................................. 142
- Records .................................................................................. 144
- Structures .............................................................................. 146
- Variable types ...................................................................... 148

### 15 Pattern matching .............................................................. 151
- Wildcard pattern ................................................................. 154
- Constant and literal patterns ............................................... 154
- Variable patterns .................................................................. 155
- Guards .................................................................................... 156
- List patterns ........................................................................... 156
- Array, record, and discriminated union patterns ................... 157
- Disjunctive and conjunctive patterns .................................... 159
- Active Pattern ...................................................................... 159
- Static and dynamic type pattern ......................................... 162
## CONTENTS

16 Higher order functions 164
   16.1 Function composition .......................... 166
   16.2 Currying ...................................... 166

17 The functional programming paradigm 169
   17.1 Functional design ............................ 170

18 Handling Errors and Exceptions 172
   18.1 Exceptions .................................. 172
   18.2 Option types ................................. 181
   18.3 Programming intermezzo: Sequential division of floats .......... 182

19 Working with files 185
   19.1 Command line arguments ....................... 186
   19.2 Interacting with the console ................... 187
   19.3 Storing and retrieving data from a file .......... 189
   19.4 Working with files and directories ............ 192
   19.5 Reading from the internet ...................... 193
   19.6 Resource Management ........................ 194
   19.7 Programming intermezzo: Ask user for existing file ......... 195

20 Classes and objects 197
   20.1 Constructors and members ..................... 197
   20.2 Accessors .................................. 199
   20.3 Objects are reference types .................. 202
   20.4 Static classes ................................ 203
   20.5 Recursive members and classes ............... 204
   20.6 Function and operator overloading ............ 205
   20.7 Additional constructors ........................ 208
   20.8 Interfacing with printf family ............... 209
   20.9 Programming intermezzo ........................ 211

21 Derived classes 215
## CONTENTS

21.1 Inheritance ............................................. 215
21.2 Abstract class ............................................ 218
21.3 Interfaces ................................................. 220
21.4 Programming intermezzo: Chess ......................... 222

22 The object-oriented programming paradigm ............. 232
  22.1 Identification of objects, behaviors, and interactions by nouns-and-verbs ................. 233
  22.2 Class diagrams in the Unified Modelling Language ................................................. 233
  22.3 Programming intermezzo: designing a racing game .................................................. 236

23 Graphical User Interfaces ................................. 241
  23.1 Opening a window ........................................ 241
  23.2 Drawing geometric primitives .......................... 243
  23.3 Programming intermezzo: Hilbert Curve .......... 252
  23.4 Handling events ......................................... 255
  23.5 Labels, buttons, and pop-up windows ............... 257
  23.6 Organising controls .................................... 263

24 The Event-driven programming paradigm ................ 269

25 Where to go from here .................................... 270

A The console in Windows, MacOS X, and Linux ........... 272
  A.1 The basics ................................................. 272
  A.2 Windows .................................................. 272
  A.3 MacOS X and Linux ...................................... 276

B Number systems on the computer ......................... 280
  B.1 Binary numbers .......................................... 280
  B.2 IEEE 754 floating point standard .................... 280

C Commonly used character sets ............................ 283
  C.1 ASCII .................................................... 283
  C.2 ISO/IEC 8859 .............................................. 283
CONTENTS

C.3 Unicode ................................................................. 284

D Common Language Infrastructure 287

E Language Details 295

E.1 Arithmetic operators on basic types 295
E.2 Basic arithmetic functions 298
E.3 Precedence and associativity 299
E.4 Lightweight Syntax 300

E To Dos 301

Bibliography 303

Index 304
1 | Preface

This book has been written as an introduction to programming for novice programmers. It is used on
the first programming course at the University of Copenhagen’s bachelor in computer science program.
It has been typeset in \LaTeX, and all programs have been developed and tested in Mono version 5.2.0.

Jon Sporring
Associate Professor, Ph.d.
Department of Computer Science,
University of Copenhagen
January 22, 2018
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 phenomena in the internet economy; and you may create programs that run on small custom-made hardware for controlling your home appliances.

2.1 Learning how to solve problems by programming

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

1. Choose a programming language: A programming language such as F# is a vocabulary and a set of grammatical rules for instructing a computer to perform a certain task. 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 teach just 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 recognise 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 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, cheap, and delivered fast, which has pulled me as a programmer in the direction of “if it works, then sell it”. On the other hand, in 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 and adapted to programming is:

Understand the problem: To solve any problem it is crucial that the problem formulation is understood: 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 find errors in and update in the future. 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 supposed to work together? Designing often involves drawing a diagram of the program, and writing program sketches 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, 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 errors, 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 fail-safe 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, 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, where the recipe emphasises what should be done in each step rather than describing the result. E.g., for making bread, you first mix yeast and water, then add flour, etc. In imperative programming
what should be done are called statements and they influence the computer’s states, like adding
flour changes the state of our dough. Almost all computer hardware is designed to execute
low-level programs written in imperative style. Imperative programming builds on the Turing
machine [10]. The first major language was FORTRAN [10] which emphasized an imperative style
of programming.

**Declarative programming**, which emphasises what a program shall accomplish but not how. We
will consider Functional programming as an example of declarative programming. A functional
programming language evaluates functions and avoids state changes. The program consists of
expressions instead of statements. As an example the function \( f(x) = x^2 \) takes a number \( x \) and
evaluates the expression \( x^2 \), and returns the result. Functional programming has its roots in
lambda calculus [11], and the first language emphasizing functional programming was Lisp [12].

**Structured programming**, which emphasises organisation of code in units with well-defined inter-
faces 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-oriented pro-
gramming 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 program-
ning language was Simula 67 developed by Dahl and Nygaard at the Norwegian Computing
Center in Oslo [2].

**Event-driven programming**, is often used when dynamically interacting with the real world. E.g.,
when programming graphical user interfaces, programs will often need to react to a user clicking
on the mouse or when text arrives from a web-server to be displayed on the screen. Event-driven
programs are often programmed using call-back functions, which are small programs that are
ready to run, when events occur.

Most programs do not follow 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 inter-
active 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. That is, F# is picky,
and will not allow the programmer to mix up types such as numbers and text. 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 contain details that are rarely used or used in contexts far from a specific programming topic. Hence, this book only includes a subset of F#, but focuses on language structures necessary to understand 4 common programming paradigms: Imperative programming mainly covered in Chapters 6 to 11, functional programming mainly covered in Chapters 13 to 16, object oriented programming in Chapters 20 and 22, and event driven programming in Chapter 23. A number of general topics are given in the appendix 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.
3 | Executing F# code

3.1 Source code

F# is a functional first programming language, meaning that it has strong support for functional programming, but F# also supports imperative and object-oriented programming. It also has strong support for parallel programming and information rich programs. It was originally developed for Microsoft’s .Net platform, but is available as open source for many operating systems through Mono. In this text, we consider F# 4.1 and its Mono implementation, which is different from .Net mainly in terms of the number of libraries accessible. The complete language specification is described in [http://fsharp.org/specs/language-spec/](http://fsharp.org/specs/language-spec/).

F# has 2 modes of execution, interactive and compiled. Interactive mode is well suited for small experiments or back-of-an-envelope calculations, but not for programming in general. Both modes can be accessed via the console, see Appendix A for more information on the console. The interactive system is started by calling fsharpi at the command prompt in the console, while compilation is performed with fsharpc, and execution of the compiled code is performed using the mono command.

F# programs comes in many forms, which are identified by suffixes. The source code is an F# program written in human readable form using an editor. F# recognises the following types of source code files:

- **.fs** An implementation file, e.g., myModule.fs
- **.fsi** A signature file, e.g., myModule.fsi
- **.fsx** A script file, e.g., gettingStartedStump.fsx
- **.fsscript** Same as .fsx, e.g., gettingStartedStump.fsscript

Compiled code is source code translated into a machine readable language, which can be executed by a machine. Compiled F# code is either:

- **.dll** A library file, e.g., myModule.dll
- **.exe** A stand-alone executable file, e.g., gettingStartedStump.exe

The implementation, signature, and script files are all typically compiled to produce an executable file, in which case they are called *scripts*, but can also be entered into the interactive system, in which case these are called *script-fragments*. The implementation and signature files are special kinds of script files used for building libraries. Libraries in F# are called modules, and they are collections of smaller programs used by other programs, which will be discussed in detail in Chapter 9.
3.2 Executing programs

Programs may either be executed by the interpreter or by compiling and executing the compiled code. In Mono the interpreter is called `fsharpi` and can be used in two ways: interactively, where a user enters one or more script-fragments separated by the “;;” characters, or to execute a script file treated as a single script-fragment.\(^1\)

To illustrate the difference between interactive and compile mode, consider the program in Listing 3.1.

```
Listing 3.1 gettingStartedStump.fsx:
A simple demonstration script.

1 let a = 3.0
2 do printfn "%g" a

The code declares a value `a` to be the decimal value 3.0 and finally prints it to the console. The `do printfn` is a statement for displaying the content of a value to the screen, and "%g" is a special notation to control how the value is printed. In this case, it is printed as a decimal number. This and more will be discussed at length in the following chapters. For now we will concentrate on how to interact with the F# interpreter and compiler.

An interactive session is obtained by starting the console, typing the `fsharpi` command, typing the lines of the program, and ending the script-fragment with “;;”. The dialogue in Listing 3.2 demonstrates the workflow. What the user types has been highlighted by a box.

```
Listing 3.2: An interactive session.

$ fsharpi
F# Interactive for F# 4.1 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
For help type #help;;

> let a = 3.0
- do printfn "%g" a;;
3
val a : float = 3.0
val it : unit = ()
>
#quit;;
```

We see that after typing `fsharpi`, then the program starts by stating details about itself followed by > indicating that it is ready to receive commands. The user then types `let a = 3.0` and presses `enter`, to which the interpreter responds with -. This indicates that the line has been received, that the script-fragment is not yet completed, and that it is ready to receive more input. When the user types `do printfn "%g" a;;` followed by `enter`, then by “;;” the interpreter knows that the script-fragment is completed, it interprets the script-fragment, responds with 3 and extra type information about the entered code, and with > to indicate, that it is ready for more script-fragments. The interpreter is

\(^1\)Jon: Too early to introduce lexeme: “F# uses many characters which at times are given special meanings, e.g., the characters “;;” is compound character denoting end of a script-fragment. Such possibly compound characters are called lexemes.”
CHAPTER 3. EXECUTING F# CODE

stopped, when the user types `#quit;;`. It is also possible to stop the interpreter by typing `ctrl-d`.

Instead of running `fsharpi` interactively, we can write the script-fragment from Listing 3.1 into a file, here called `gettingStartedStump.fsx`. This file can be interpreted directly by `fsharpi` as shown in Listing 3.3.

Listing 3.3: Using the interpreter to execute a script.

```
$ (fsharpi gettingStartedStump.fsx)
```

Notice that in the file, `;;` is optional. We see that the interpreter executes the code and prints the result on screen without the extra type information.

Finally, the file containing Listing 3.1 may be compiled into an executable file with the program `fsharpc`, and run using the program `mono` from the console. This is demonstrated in Listing 3.4.

Listing 3.4: Compiling and executing a script.

```
$ (fsharpc gettingStartedStump.fsx)
```

The compiler takes `gettingStartedStump.fsx` and produces `gettingStarted.exe`, which can be run using `mono`.

Both the interpreter and the compiler translates the source code into a format, which can be executed by the computer. While the compiler performs this translation once and stores the result in the executable file, the interpreter translates the code every time the code is executed. Thus, to run the program again with the interpreter, it must be retranslated as "$fsharpi gettingStartedStump.fsx". In contrast, compiled code does not need to be recompiled to be run again, only re-executed using "$ mono gettingStartedStump.exe". On a MacBook Pro, with a 2.9 Ghz Intel Core i5, the time the various stages take for this script are:

```
<table>
<thead>
<tr>
<th>Command</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>fsharpi gettingStartedStump.fsx</td>
<td>1.88s</td>
</tr>
<tr>
<td>fsharpc gettingStartedStump.fsx</td>
<td>1.90s</td>
</tr>
<tr>
<td>mono gettingStartedStump.exe</td>
<td>0.05s</td>
</tr>
</tbody>
</table>
```

I.e., executing the script with `fsharpi` is slightly faster than by first compiling it with `fsharpc` and then executing the result with `mono`, 1.88s $< 0.05s + 1.90s$, if the script were to be executed only once, but every future execution of the script using the compiled version requires only the use of `mono`, which is much faster than `fsharpi`, 1.88s $\gg$ 0.05s.

The interactive session results in extra output on the **type inference** performed, which is very useful for **debugging** and development of code-fragments, but both executing programs with the interpreted directly from a file and compiling and executing the program is much preferred for programming complete programs, since the starting state is well defined, and since this better supports **unit-testing**, which is a method for debugging programs. Thus, **prefer compiling over interpretation**.
Programming is the art of solving problems by writing a program to be executed by a computer. For example, to solve the following problem,

**Problem 4.1**

What is the sum of 357 and 864?

we have written the program in F# shown in Listing 4.1.

```fsharp
1 let a = 357
2 let b = 864
3 let c = a + b
4 do printfn "%d" c
```

In box the above, we see our program was saved as a script in a file called `quickStartSum.fsx`, and in the console we executed the program by typing the command `fsharpc --nologo quickStartSum.fsx && mono quickStartSum.exe`. The result is then printed in the console to be 1221. Here, as in the rest of this book, we have used the optional flag --nologo, which informs fsharpc not to print information about its version etc., thus making the output shorter. The `&&` notation tells the console to first run the command on the left, and if that did not report any errors, then run that on the right. This could as well have been performed as two separate commands to the console, and throughout this book, we will use the above shorthand, when convenient.

To solve the problem, we made program consisting of several lines, where each line was an expression. The first expression `let a = 357` in line 1 used the `let` keyword to bind the value 357 to the name `a`. This is called a let-binding, and a let-binding makes the name synonymous with the value. Another point to be noted is that F# identifies 357 as an integer number, which is F#'s preferred number type, since computations on integers are very efficient, and since integers are very easy to communicate to other programs. In line 2 we bound the value 864 to the name `b`, and to the name `c`, we bound the result of evaluating the sum `a + b` in line 3. Line 4 is a do-binding, as noted by the keyword `do`. Do-bindings are also sometimes called statements, and the `do` keyword is optional in F#. Here the value of `c` was printed to the console followed by a newline (LF possibly preceded by CR, see Appendix C.1) with the `printfn` function. A function in F# is an entity that takes zero or more arguments and returns a value. The function `printfn` is very special, since it can take any number of arguments. It need not return any value, but F# insists that every function must return a value, wherefore `printfn`
returns a special type of value called **unit** and written as “()”. The do tells F# to ignore this value. Here **printfn** has been used with 2 arguments: "%A" and c. Notice that in contrast to many other languages, F# does not use parentheses to frame the list of arguments, nor does it use commas to separate them. In general, the **printfn** function always has 1 or more arguments, and the first is a *format string*. A *string* is a sequence of characters starting and ending with double quotation marks. E.g., let s = "this is a string of characters" binds the string "this is..." to the name s. For the **printfn** function, the format string may be any string, but if it contains format character sequences, such as %A, then the values following the format string are substituted. The format string must match the value *type*, that is, here c is of type integer, whereas the format string %A matches many types.

Types are a central concept in F#. In the script 4.1 we bound values of integer type to names. There are several different integer types in F#, here we used the one called **int**. The values were not declared to have these types, instead the types were inferred by F#. Typing these bindings line by line in an interactive session, then we see the inferred types as shown in Listing 4.2.

---

**Listing 4.2: Inferred types are given as part of the response from the interpreter.**

```fsharp
1 > let a = 357;;
2 val a : int = 357
3
4 > let b = 864;;
5 val b : int = 864
6
7 > let c = a + b;;
8 val c : int = 1221
9
10 > do printfn "%A" c;;
11 1221
12 val it : unit = ()
```

---

The interactive session displays the type using the *val* keyword followed by the name used in the binding, its type, and its value. Since the value is also responded, then the last **printfn** statement is superfluous. However, *it is ill advised to design programs to be run in an interactive session*, since the scripts needs to be manually copied every time it is to be run, and since the starting state may be unclear. Notice that **printfn** is automatically bound to the name *it* of type **unit** and value “()”. F# insists on binding all statements to values, and in lack of an explicit name, then it will use it. Rumor has it that it is an abbreviation for "irrelevant".

Were we to solve a slightly different problem,

---

**Problem 4.2**

What is the sum of 357.6 and 863.4?

---

where the only difference is that the numbers now use a *decimal point*. These are called *floating point numbers*, and the internal representation is quite different to integer numbers used previously, and the algorithms used to perform arithmetic are also quite different from integers. Now the program would look like Listing 4.3.
CHAPTER 4. QUICK-START GUIDE

Listing 4.3 quickStartSumFloat.fsx:
Floating point types and arithmetic.

1 let a = 357.6
2 let b = 863.4
3 let c = a + b
4 do printfn "%A" c

1 $fsharpc --nologo quickStartSumFloat.fsx & mono quickStartSumFloat.exe
2 1221.0

On the surface, this could appear as an almost negligible change, but the set of integers and the set of real numbers (floats) require quite different representations, in order to be effective on a computer, and as a consequence, the implementation of their operations such as addition are very different. Thus, although the response is an integer, it has type `float`, which is indicated by `1221.0`, and which is not the same as `1221`. F# is very picky about types, and generally does not allow types to be mixed, as demonstrated in the interactive session in Listing 4.4.

Listing 4.4: Mixing types is often not allowed.

1 > let a = 357;;
2 val a : int = 357
3
4 > let b = 863.4;;
5 val b : float = 863.4
6
7 > let c = a + b;;
8 let c = a + b;;
9 --------------
10 stdin(4,13): error FS0001: The type 'float' does not match the type 'int'

we see that binding a name to a number without a decimal point is inferred to be integer, while when binding to a number with a decimal point, then the type is inferred to be a float, and when trying to add values of integer and floating point, we get an error. The error message contains much information. First it states that the error is in stdin(4,13), which means that the error was found on standard-input at line 4 and column 13. Since the program was executed using `fsharpi quickStartSumFloat.fsx`, then here standard input means the file `quickStartSumFloat.fsx` shown in Listing 4.3. The corresponding line and column is also shown in Listing 4.4. After the file, line, and column number, F# informs us of the error number, and a description of the error. Error numbers are an underdeveloped feature in Mono, and should be ignored. However, the verbal description often contains useful information for debugging. In the example we are informed that there is a type mismatch in the expression, i.e., since `a` is an integer, then F# had expected `b` to be one too. Debugging is the process of solving errors in programs, and here we can solve the error by either making `a` into a float or `b` into an int. The right solution depends on the application.

F# is a functional first programming language, and one implication of this is that names have a lexical scope. A scope are the lines in a program, where a binding is valid, and lexical scope means that to find the value of a name F# looks for the value in the above lines. Further, at the outer most level, rebinding is not allowed. If attempted, then F# will return an error as shown in Listing 4.5.
However, if the same is performed in an interactive session, then rebinding does not cause an error as shown in Listing 4.6.

Listing 4.6: Names may be reused when separated by the lexeme “;;”.

```
> let a = 357;;
val a : int = 357
> let a = 864;;
val a : int = 864
```

The difference is that the “;;” lexeme is used to specifies the end of a script-fragment. A lexeme is a letter or a word, which the F# considers as an atomic unit. Script-fragments may be defined both in scripts and in interactive mode, and rebinding is not allowed at the outermost level in script-fragments. Even with the “;;” lexeme, rebinding is not allowed in compile-mode. In general, avoid rebinding of names.

In F#, functions are also values, and we may define a function `sum` as part of the solution to the above program as shown in Listing 4.7.

Listing 4.7 quickStartSumFct.fsx:
A script to add 2 numbers using a user defined function.

```
let sum x y = x + y
let c = sum 357 864
do printfn "%A" c
```

Functions are useful to encapsulate code, such that we can focus on the transformation of data by a function while ignore the details on how this is done. Functions are also useful for code reuse, i.e., instead of repeating a piece of code in several places, such code can be encapsulated in a function and replaced with function calls. This makes debugging and maintenance considerably simpler. Entering the function into an interactive session will illustrate the inferred type, the function `sum` has:

```
val sum : x:int -> y:int -> int
```

The “->” is the mapping operator in the sense that functions are mappings between sets. The type of the function `sum`, should be read as `val sum : x:int -> (y:int -> int)`, that is, `sum` takes an integer and returns a function, which takes an integer and returns an integer. This is an example of a higher-order function.
Type inference in F# may cause problems, since the type of a function is inferred in the context, in which it is defined. E.g., in an interactive session, defining the sum in one scope on a single line will default the types to integers, F#’s favorite type. Thus, if the next script-fragment uses the function with floats, then we will get an error message as shown in Listing 4.8.

Listing 4.8: Types are inferred in blocks, and F# tends to prefer integers.

```fsharp
val sum : x:int -> y:int -> int
> let c = sum 357.6 863.4;;
let c = sum 357.6 863.4;;

stdin(3,13): error FS0001: This expression was expected to have type
'int'
but here has type
'float'
```

A remedy is to define the function in the same script-fragment as it is used such as shown in Listing 4.9.

Listing 4.9: Type inference is per script-fragment.

```fsharp
> let sum x y = x + y
- let c = sum 357.6 863.4;;
val sum : x:float -> y:float -> float
val c : float = 1221.0
```

Alternatively, the types may be explicitly stated as shown in Listing 4.10.

Listing 4.10: Function argument and return types may be stated explicitly.

```fsharp
> let sum (x : float) (y : float) : float = x + y;;
val sum : x:float -> y:float -> float
> let c = sum 357.6 863.4;;
val c : float = 1221.0
```

The function `sum` has two arguments and a return type, and in Listing 4.10 we have specified all three. This is done using the “;” lexeme, and to resolve confusion, we must use parentheses around the arguments such as `(y : float)`, otherwise F# would not be able to understand, whether the type annotation was for the argument or the return value. Often it is sufficient to specify some of the types, since type inference will enforce the remaining types. E.g., in this example, the “+” operator is defined for identical types, so specifying the return value of `sum` to be a float, implies that the result of the “+” operator is a float, and therefore its arguments must be floats, and finally then the arguments for `sum` must be floats. However, in this book we advocate the following advice: **specify types unless explicitly working with generic functions.**

In this chapter, we have scratched the surface of learning how to program by concentrating on a number of key programming concepts and how they are expressed in the F# language. In the following chapters, we will expand the description of F# with features used in all programming approaches.