Chapter 1

Solving problems by writing a program

Abstract In this chapter, you will find a quick introduction to several essential programming constructs with several examples that you can try on your computer using the `dotnet` command in your console. All constructs will be discussed in further detail in the following chapters. In this chapter, you will get a peek at:

- How to execute an F# program.
- How to perform simple arithmetic using F#.
- What types are and why they are important.
- How to write to and obtain written input from the user.
- How to perform conditional execution of code.
- How to define functions.
- How to repeat code without having to rewrite them.
- How to add textual comments to help yourself and other programmers understand your programs.

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 1.1

What is the sum of 357 and 864?

we have written the program shown in Listing 1.1. In this book, we will show many

Listing 1.1 quickStartSum.fsx:
A script to add 2 numbers and print the result to the console.

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

```
$ dotnet fsi quickStartSum.fsx
1221
```

programs, and for most, we will also show the result of executing the programs on a computer. Listing 1.1 shows both our program and how this program is executed on a computer. In the listing, we see our program was saved as a script in a file called `quickStartSum.fsx`, and in the console (also known as the terminal and the command-line) we executed the program by typing the command `dotnet fsi quickStartSum.fsx`. The result is then printed by the computer to the console as `1221`. The colors are not part of the program but have been added to make it easier for us to identify different syntactical elements of the program.

The program consists of several lines. Our listing shows line numbers to the left. These are not part of the program but added for ease of discussion, since the order in which the lines appear the program matters. In this program, each line contains *expressions*, and this program has `let`-, `do`-expressions, and an addition. `let`-expressions defines aliases, and `do`-expressions defines computations. `let` and `do` are examples of *keywords*, and "+" is an example of an *operator*. Keywords, operators, and other sequences of characters, which F# recognizes are jointly called *lexemes*.

Reading the program from line 1, the first expression we encounter is `let a = 357`. This is known as a *let-binding* in F# and defines the equivalence between the name `a` and the value `357`. F# does not accept a keyword as a name in a `let`-bindings. The consequence of this line is that in later lines there is no difference between writing the name `a` and the value `357`. Similarly in line 2 the value 864 is bound to the name `b`. In contrast, line 3 contains an addition and a `let`-expression. It is at times useful to simulate the execution the computer does in a step-by-step manner by replacement:

```
let c = a + b  ⇝  let c = 357 + 864  ⇝  let c = 1221
```

Thus, since the expression on the right-hand side of the equal sign is evaluated, the result of line 3 is that the name `c` is bound to the value `1221`.

Line 4 has a `do`-expression is also called a *do-binding* or a *statements*. In this `do`-binding, the *`printfn` function* `printfn` is called with 2 arguments, `"%A"`and `c`. All functions return values, and `printfn` the value 'nothing', which is denoted *"`()`"*. This function is very commonly used but also very special since it can take any number of arguments and produces output to the console. We say that "the output is printed to the screen". The first argument is called the *formatting string* and describes, what should be printed and how the remaining arguments if any, should be formatted. In this case, the value `c` is printed as an integer followed by a newline. Notice that in contrast to many other languages, F# does not use parentheses to frame the list of function arguments, nor does it use commas to separate them.

1.1 Executing F# programs on a computer

The main purpose of writing programs is to make computers execute or run them. F# has two modes of execution, *interactive* and *compiled*. Interactive mode allows the user to interact with F# as a dialogue: The user writes statements, and F# responds immediately. If a program has been saved as a file as in Listing 1.1 we do not need to rewrite the complete program every time we wish to execute it but can give the file as input to the F#'s interactive mode as demonstrated in Listing 1.1. Interactive mode is well suited for small experiments or back-of-an-envelope calculations, but not for programming in general, since each line is interpreted anew every time the program is run. In contrast, in compile mode, dotnet interprets the content of a source file once, and writes the result to disk, such that every when the user wishes to run the program, the interpretation step is not performed. For large programs, this can save considerable time. In the first chapters of this book, we will use interactive mode, and compile mode will be discussed in further detail in **??**.

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 1.2 demonstrates the workflow. What the user types has been highlighted by a box.

Listing 1.2: An interactive session.

```
$ dotnet fsi

Microsoft (R) F# Interactive version 12.0.0.0 for F# 6.0
Copyright (c) Microsoft Corporation. All Rights Reserved.

For help type #help;;

> let a = 3
- do printfn "%A" a;;
3
val a : int = 3
val it : unit = ()

> #quit;;
```

We see that after typing `fsharpi`, the program starts by stating details about itself. Then F# writes > indicating that it is ready to receive commands. The user types `let a = 3` 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 "%A" a;;` followed by `enter`, then by "`;;`" the interpreter knows that the script-fragment is completed, it interprets the script-fragment, responds with `3` and some extra information about the entered code, and with > to indicate that it is ready for more script-fragments. The interpreter is stopped when the user types `#quit;;`. It is also possible to stop the interpreter by typing `ctrl-d`.

The interactive session results in extra output on the *type inference* performed. In Listing 1.2, F# states that the name `a` has *type* `int` and the value `3`. Likewise, the `do` statement F# refers to by the name `it`, and it has the type `unit` and value "`()`". Types are very important to F# since they define how different program pieces fit together like lego bricks. They are a key ingredient for finding errors in programs, also known as *debugging*, and much of the rest of this book is concerned with types.

Instead of running `fsharpi` interactively, we can write the script-fragment from Listing 1.2 into a file, here called `gettingStartedStump.fsx`. This file can be interpreted directly by `dotnet fsi` as shown in Listing 1.3.

Listing 1.3: Using the interpreter to execute a script.

```
$ dotnet fsi gettingStartedStump.fsx
3
```

Notice that in the file, "`;;`" is optional. In comparison to Listing 1.2, we see that the interpreter executes the code and prints the result on screen without the extra type information.

Files are important when programming, and F# and the console interprets files differently by the filename's suffix. A filename's suffix is the sequence of letters after the period in the filename. Generally, there are two types of files: *source code* and compiled programs. Until **??**, we will concentrate on script files, which are source code, written in human-readable form using an editor, and has `.fsx` or `.fsscript` as suffix. In Table 1.1 is a complete list of possible suffixes used by F#.

Suffix	Human readable	Description
`.fs`	Yes	An *implementation file*, e.g., `myModule.fs`
`.fsi`	Yes	A *signature file*, e.g., `myModule.fsi`
`.fsx`	Yes	A *script file*, e.g., `gettingStartedStump.fsx`
`.fsscript`	Yes	Same as `.fsx`, e.g., `gettingStartedStump.fsscript`
`.dll`	No	A *library file*, e.g., `myModule.dll`
`.exe`	No	A stand-alone *executable file*, e.g., `gettingStartedStump.exe`

Table 1.1 Suffixes used, when programming F#.

1.2 Values have types and types reduce the risk of programming errors

Types are a central concept in F#. In the script 1.1 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, we see the inferred types as shown in Listing 1.4. The interactive session displays the type using the

Listing 1.4: Inferred types are given as part of the response from the interpreter.

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

> let b = 864;;
val b: int = 864

> let c = a + b;;
val c: int = 1221

> do printfn "%A" c;;
1221
val it: unit = ()
```

val keyword followed by the name used in the binding, its type, and its value. Since the value is also returned, the last `printfn` statement is superfluous. 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, it will use `it`. Rumor has it that `it` is an abbreviation for "irrelevant".

In mathematics, types are also an important concept. For example, a number may belong to the set of natural $\mathbb{N}$, integer $\mathbb{Z}$, or real numbers, where all 3 sets are infinitely large and $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{R}$ as illustrated in Figure 1.1. For many problems,

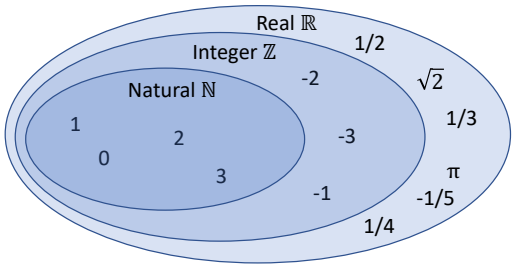

Fig. 1.1 In mathematics, the sets of natural, integer, and real numbers are each infinitely large, and real contains integers which in turn contains the set of natural numbers.

working with infinite sets is impractical, and instead, a lot of work in the early days of the computer's history was spent on designing finite sets of numbers, which have many of the properties of their mathematical equivalent, but which also are efficient for performing calculations on a computer. For example, the set of integers in F# is called `int` and is the set $\{-2\,147\,483\,648 \ldots 2\,147\,483\,647\}$.

Types are also important when programming. For example, a the `string` `"863"` and the `int` `863` may conceptually be identical but they are very different in the computer. F# is very picky about types, and generally does not allow types to be mixed, as demonstrated in the interactive session in Listing 1.5. In this ex-

Listing 1.5: Mixing types is often not allowed.

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

> let b = "863";;
val b: string = "863"

> let c = a + b;;

  let c = a + b;;
  ------------^

/Users/jrh630/repositories/fsharp-book/src/stdin(3,13): error
   FS0001: The type 'string' does not match the type 'int'
```

ample, we see that adding a string to an integer results in an error. The *error message* contains much information. First, it illustrates where dotnet could not understand the input by `------------^`. Then it repeats where the error is found as `/User/.../src/stdin(3,13)`, which means that `dotnet` was started in the directory `/User/.../src/`, the input was given on the standard-input meaning the keyboard, and the error was detected on line 3, column 13. Then F# gives the error number and a description of the error. Error numbers are an underdeveloped feature

in F# and should be ignored. However, the verbal description often contains useful information for correcting the program. Here, we are informed that there is a type mismatch in the expression. The reason for the mismatch is that since `a` is an integer, then the "+" operator must be integer addition, and thus for the expression to be executable, `b` can only be an integer.

1.3 Organizing often used code in functions

`printfn` is an example of a built-in function, and very often we wish to define our own. For example, in longer programs, some code needs to be used in several places, and defining functions to *encapsulate* such code can be a great advantage for reducing the length of code, debugging, and writing code, which is easier to understand by other programmers. A function is defined using a `let`-binding. For example, to define a function, which takes two integers as input and returns their sum, we write

Listing 1.6: Defining the function `sum`

```
1 let sum x y =
2   x + y
```

What this means is that we bind the name `sum` as a function, which takes two arguments and adds them. Further, in the function, the arguments are locally referred to by the names `x` and `y`. Indentation determines which lines should be evaluated when the function is called, and in this case, there is only one. The value of the last expression evaluated in a function is its return value. Here there is only one expression `x+y`, and thus, this function returns the value of the addition. This program does not do anything, since the function is neither called nor is its output used. However, we can modify Listing 1.1 to include it as shown in Listing 1.7. The output is the

Listing 1.7 quickStartSumFct.fsx:
Adding two integers with the use of a in-code defined function.

```
1 let sum x y =
2   x + y
3 let c = sum 357 864
4 do printfn "%A" c
```

```
1 $ dotnet fsi quickStartSumFct.fsx
2 1221
```

same for the two programs, and the computation performed is almost the same. A step-by-step manner by replacement of the computation performed in line 3 is

```
let c = sum 357 864   ⇝   let c = 357 + 864   ⇝   let c = 1221
```

The main difference is that with the function `sum` we have an independent unit, which can be reused elsewhere in the code.

1.4 Asking the user for input

The `printfn` function allows us to write to the screen, which is useful, but sometimes we wish to start a dialogue with the user. One way to get user input is to ask the user to type something on the keyboard. Technically, input from the keyboard is called an *`stdin` stream*. This terminology is intended to remind us of characters streaming from the keyboard like the flow of water in a stream. Computer streams are different than water streams in that characters (or other items) only flow, when we ask for them. F# provides many libraries of prebuilt functions, and here we will use the `System.Console.ReadLine` function. The "`.`"-lexeme is read as `ReadLine` is a function which lies in `Console` which in turn lies in `System`. In the function documentation, we can read that `System.Console.ReadLine` takes a unit value as an argument and returns the *`string`* the user typed. A string is a built-in type, as is an integer, and strings contain sequences of characters. The program will not advance until the user presses the newline. An example of a program that multiplies two integers supplied by a user is given in Listing 1.8, In this program, we find a user

Listing 1.8 quickStartSumInput.fsx:
Asking the user for input. The user entered 6, pressed the return button, 2, and pressed return again.

```
let sum x y = x + y
printfn "Adding a and b"
printf "Enter a: "
let a = int (System.Console.ReadLine ())
printf "Enter b: "
let b = int (System.Console.ReadLine ())
let c = sum a b
do printfn "%A" c
```

```
dotnet fsi quickStartSumInput.fsx
Adding a and b
Enter a: 6
Enter b: 2
8
```

dialogue, and we have designed it such that we assume that the user is unfamiliar with the inner workings of our program, and therefore helps the user understand the

purpose of the input and the expected result. This is good programming practice. Here, we will not discuss the program line-to-line, but it is advised to the novice programmer to match what is printed on the screen and from where in the code, the output comes from. However, let us focus on line 4 and 4, which introduce two new programming constructs. In each of these lines, 3 things happen: First the System.Console.ReadLine function is called with the "`()`" value as argument. This reads all the characters, the user types, up until the user presses the return key. The return value is a string of characters such as `"6"`. This value is different from the integer 6, and hence, to later be able to perform integer-addition, we *cast* the string value to `int`, meaning that we call the function `int` to convert the string-value to the corresponding integer value. Finally, the result is bound to the names `a` and `b` respectively.

1.5 Conditionally execute code

Often problem requires code evaluated based on conditions, which only can be decided at *runtime*, i.e., at the time, when the program is run. Consider a slight modification of our problem as

Problem 1.2

Ask for two integer values from the user, a and b, and print the result of the integer division a/b.

To solve this problem, we must decide what to do, if the user inputs $b = 0$, since division by zero is ill-defined. This is an example of a user input error, and later, we will investigate many different methods for handling such errors, but here, we will simply write an error message to the user, if the desired division is ill-defined. Thus, we need to decide at *runtime*, whether to divide a and b or to write an error message. For this we will use the `match-with` expression. In this program, the `match-with` expression covers line 7 to 12. When the computer executes these lines, it checks the value `b` against a list of patterns separated by "`|`". The code belonging to each pattern follows the arrow, "`->`", and is called a *branch*, and which lines belong to each branch is determined by *indentation*. Hence, the code belonging to the "`0.0`" branch is line 9 and to the "`_`" branch is line 11 to 12. The branches are checked one at a time from top to bottom. I.e., first `b` is compared with `0` which is equivalent to check whether `b = 0.0`. If this is true, then its branch is executed. Otherwise, the `b` is compared with the *wildcard* pattern, "`_`". The wildcard pattern matches anything, and hence, if `b` is nonzero, then "`_`"-branch is executed. In most cases, F# will give an error, if the list of patterns does not cover the full domain of the type being matched. Here, `b` is an integer, and thus, we must write branches that take *all* integer values into account. The wildcard pattern makes this easy and works as a catch-all-other case, and is often placed as the last case, following the important cases. Assuming

Listing 1.9 quickStartDivisionInput.fsx: Conditionally divide two user-given values.

```
let div x y = x / y
printfn "Dividing a by b"
printf "Enter a: "
let a = int (System.Console.ReadLine ())
printf "Enter b: "
let b = int (System.Console.ReadLine ())
match b with
  0 ->
    do printfn "Input error: Cannot divide by zero"
  | _ ->
    let c = div a b
    do printfn "%A" c
```

```
% dotnet fsi quickStartDivisionInput.fsx
Dividing a by b
Enter a: 6
Enter b: 2
3
% dotnet fsi quickStartDivisionInput.fsx
Dividing a by b
Enter a: 6
Enter b: 0
Input error: Cannot divide by zero
```

that the user enters the value 0, then the step-by-step simplification of `match-with` expression is,

```
match b  with 0 -> ... | _ -> ...
⇝  do printfn "Input error: Cannot divide by zero"
```

1.6 Repeatedly execute code

Often code needs to be evaluated many times or looped. For example, instead of stopping the program in Listing 1.9 if the user inputs $b = 0$, then we could repeat the question as many times as needed until the user inputs a non-zero value for b. This is called a loop, and there are several programming constructions for this purpose.

Let us first consider recursion. A recursive function is one, which calls itself, e.g., $f(f(f(\ldots(x))))$ is an example of a function f which calls itself many times, possibly infinitely many. In the latter case, we say that the recursion has entered an infinite loop, and we will experience that either the program runs forever or that the execution stops due to a memory error. If we had infinite memory. To avoid this,

recursive functions must always have a stopping criterion. Thus, we can design a function for asking the user for a non-zero input value as shown in Listing 1.10. The function `readNonZeroValue` takes no input denoted by "`()`", and repeatedly

Listing 1.10 quickStartRecursiveInput.fsx:
Recursively call `ReadLine` until a non-zero value is entered.

```
1  let rec readNonZeroValue () =
2    let a = int (System.Console.ReadLine ())
3    match a with
4      0 ->
5        printfn "Error: zero value entered. Try again"
6        readNonZeroValue ()
7      | _ ->
8        a
9  printfn "Please enter a non-zero value"
10 let b = readNonZeroValue ()
11 printfn "You typed: %A" b
```

```
1  $ dotnet fsi quickStartRecursiveInput.fsx
2  Please enter a non-zero value
3  0
4  Error: zero value entered. Try again
5  3
6  You typed: 3
```

calls itself until the $a \neq 0$ condition is met. It is recursive since its body contains a call to itself. For technical reasons, F# requires recursive functions to be declared by the *rec*-keyword as demonstrated. The function has been designed to stop if $a \neq 0$, and in F#, this is tested with the "<>" operator. Thus, if the stopping condition is satisfied, then the `then`-branch is executed, which does not call itself, and thus the recursion goes no deeper. If the condition is not met, then the `else`-branch is executed, and the function is eventually called anew. The example execution of the program demonstrates this for the case that the user first inputs the value 0 and then the value 3.

As an alternative to recursive functions, loops may also be implemented using the *while*-expression. In Listing 1.10 is an example of a solution where the recursive loop has been replaced with `while`-loop. As for other constructs, the lines to be repeated are indicated by indentation, in this case, lines 4 to 5, and in the end, the result of the `readNonZeroValueAlt` function is the last expression evaluated, which is the trivial expression `a` in line 6. In comparison with the recursive version of the program, the `while`-loop has a continuation conditions (line 3), i.e., the content of the loop is repeated as long as `a = 0` evaluates to `true`. Another difference is that in Listing 1.10 we could simplify our program to only using `let` value-bindings, here we need a new concept: *variables* also known as a *mutable* value. Mutable values allow us to update the value associated with a given name. Thus, the value associated with a name of mutable type depends on when it is accessed.

Listing 1.11 quickStartWhileInput.fsx:
Replacing recursion in Listing 1.10 with a `while`-loop.

```
1 let readNonZeroValueAlt () =
2   let mutable a = int (System.Console.ReadLine ())
3   while a = 0 do
4     printfn "Error: zero value entered. Try again"
5     a <- int (System.Console.ReadLine ())
6   a
7 printfn "Please enter a non-zero value"
8 let b = readNonZeroValueAlt ()
9 printfn "You typed: %A" b
```

```
1 $ dotnet fsi quickStartWhileInput.fsx
2 Please enter a non-zero value
3 0
4 Error: zero value entered. Try again
5 3
6 You typed: 3.0
```

This construction makes programs much more complicated and error-prone, and their use should be minimized. The syntax of mutable values is that first it should be defined with the `mutable`-keyword as shown in line 2, and when its value is to be updated then the "<-"-notation must be used as demonstrated in line 5. Note that the execution of the two programs Listing 1.10 and Listing 1.11, gives identical output, when presented with identical input. Hence, they solve the same problem by two quite different means. This is a common property of solutions to problems as a program: Often several different solutions exist, which are identical on the surface, but where the quality of the solution depends on how quality is defined and which programming constructions have been used. Here, the main difference is that the recursive solution avoids the use of mutable values, which turns out to be better for proving the correctness of programs and for adapting programs to super-computer architectures. However, recursive solutions may be very memory intensive, if the recursive call is anywhere but the last line of the function.

1.7 Programming as a form of communication

When programming it is important to consider the time dimension of a program. Some usually very small programs are only used for a short while, e.g., to test a programming construction or an idea to a solution. Others small as well as large may be used again and again over a long period, and possibly given to other programmers to use, maintain, and extend. In this case, programming is an act of communication, where what is being communicated is the solution to a problem as well as

the thoughts behind the chosen solution. Common experiences among programmers are that it is difficult to fully understand the thoughts behind a program written by a fellow programmer from its source code alone, and for code written perhaps just weeks earlier by the same programmer, said programmer can find it difficult to remember the reasons for specific programming choices. To support this communication, programmers use code-*comments*. As a general concept, this is also called in-code documentation. Documentation may also be an accompanying manual or report. Documentation serves several purposes:

1. Communicate what the code should be doing, e.g., describe functions in terms of their input-output relation.
2. Highlight big insights essential for the code.
3. Highlight possible conflicts and/or areas where the code could be changed later.

F# has two different syntaxes for comments. A block comment is everything bracketed by `(* *)`, and a line comment, is everything between "`//`" and the end of the line. For example, adding comments to Listing 1.10 could look like Listing 1.12 Comments are ignored by the computer and serve solely as programmer-

Listing 1.12 quickStartRecursiveInputComments.fsx: Adding comments to Listing 1.10.

```
(*
  Demonstration of recursion for keyboard input.
  Author: Jon Sporring
  Date: 2022/7/28
*)

// Description: Repeatedly ask the user for a non-zero number
//   until a non-zero value is entered.
// Arguments: None
// Result: the non-zero value entered
let rec readNonZeroValue () =
  // Note that the value of a is different for every
  // recursive call.
  let a = int (System.Console.ReadLine ())
  match a with
    0 ->
      printfn "Error: zero value entered. Try again"
      readNonZeroValue ()
    | _ ->
      a
printfn "Please enter a non-zero value"
let b = readNonZeroValue ()
printfn "You typed: %A" b
```

to-programmer communication, there are no or few rules for specifying, what is good and bad documentation of a program. The essential point is that coding is a

journey in problem-solving, and proper documentation is an aid in understanding the solution and the journey that lead to it.

1.8 Key Concepts and Terms in This Chapter

- F# has two modes of operation: **Interactive** and **compile** mode. The first chapters of this book will focus on the interactive mode.
- F# is accessed through the **console/terminal/command-line**, which is another program, in which text commands can be given such as starting the dotnet program in interactive mode.
- Programs are written in a human-readable form called the **source-code**.
- Source code consists of several syntactical elements such as **operators** such as "*" and "<-", **keywords** such as "let" and "while", **values** such as 1.2 and the string "hello world", and **user-defined names** such as "a" and "str". All words, which F# recognizes are called **lexemes**.
- A program consists of a sequence of **expressions**, which comes in two types: **let** and **do**.
- Values have **types** such as **int** and **string**. When performing calculations, the type defines which calculations can be done.
- **Functions** are a type of value and defined using a let-binding. They are used to encapsulate code to make the code easier to read and understand and to make code reusable.
- The **conditional match-with** expression is used to control what code is to be executed at **runtime**. Each piece of conditional code is called a **branch**
- **Recursion** and **while**-loops are programming structure to execute the same code several times.
- **Mutable values** are in contrast to **immutable values** may change value over time, and makes programmer harder to understand.
- **Comments** are **in-code documentation** and are ignored by the computer but serve as an important tool for communication between programmers.