Jon Sporring

Department of Computer Science,
University of Copenhagen

Learning to Program with F#

2023-09-18

Springer Nature

Preface

This book has been written as an introduction to programming for novice programmers. It is used in 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 dotnet version 6.0.101

Jon Sporring
Professor, Ph.d.
Department of Computer Science,
University of Copenhagen
2023-09-18

Contents

Chapter 1

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 interact with other users, databases, and artificial intelligence; you may create programs that run on supercomputers 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.

1.1 How to Learn to Solve Problems by Programming

To learn how to program, there are a couple of useful steps:

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 still be expressed in some fairly rigorous way. Theoretical computer scientists typically do not rely on computers or programming languages but use mathematics to prove the properties of algorithms. However, most computer scientists program using a computer, 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 express 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 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 internet and through other works you will be able to learn much more.

3. Practice: To be a good programmer, the most essential step is: practice, practice, practice! It has been estimated that to master anything, then you have to have spent at least 10000 hours practicing, so get started logging hours! It of course matters, how you practice. This book teaches several different programming themes. The point is that programming is thinking, and the scaffold you use shapes your thoughts. It is therefore important to recognize this scaffold and to have the ability to choose one which suits your ideas and your goals best. The best way to expand your abilities is to sharpen your present abilities, push yourself into new territory, and try 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 specific solutions, has forced me to put the programming tools and techniques that I use into perspective. Sometimes a task requires a cheap and fast solution, other times customers want a long-perspective solution with bug fixes, upgrades, and new features. Practicing solving real problems helps you strike a balance between the two when programming. It also allows makes you a more practical programmer, by allowing you to recognize its applications in your everyday experiences. Regardless, real problems create real programmers.

1.2 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, 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. 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 lead to programs that are faster to implement, easier to find errors in, and easier to update in the future. 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. Furthermore, the solution to many problems will have several 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 how long time they take to run on a computer. 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 require rethinking the design. Most often the implementation step also requires careful documentation of key aspects of the code, e.g., a user manual for the user, and internal notes for fellow programmers that are to maintain and update the code in the future.

Reflect on the result: A crucial part of any programming task is ensuring that the program solves the problem sufficiently. Ask yourself questions such as: 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? Which parts of your method would you avoid or replicate in future programming sessions? Can you reuse some of the code you developed in other programs?

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.

1.3 Approaches to Programming

This book focuses on several fundamentally different approaches to programming:

Declarative programming emphasizes *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, evaluates the expression x^2, and returns the resulting number. Everything about the function may be

characterized by the relation between the input and output values. Functional programming has its roots in lambda calculus [1]. The first language emphasizing functional programming was Lisp [7].

Imperative programming emphasizes *how a program shall accomplish a solution* and focusses less on *what the solution is*. A cooking recipe is an example of the spirit of imperative programming, where the recipe emphasizes what should be done in each step rather than describing the result. For example, a bread recipe might tell you to first mix yeast and water, then add flour, etc. In imperative programming what should be done is called *statements* and in the recipe analogy, the steps are the statements. Statements influence the computer's *states*, in the same way, that 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]. As a historical note, the first major language was FORTRAN [6] which emphasized an imperative style of programming.

Structured programming emphasizes organization of programs in units of code and data. For example, a traffic light may consist of a state (red, yellow, green), and code for updating the state, i.e., switching from one color to the next. We will focus on Object-oriented programming as an example of structured programming. *Object-oriented programming* is a type of programming, where the code and data are structured into *objects*. E.g., a traffic light may be an object in a traffic-routing program. The first object-oriented programming language was Simula 67 developed by Dahl and Nygaard at the Norwegian Computing Center in Oslo [2].

Event-driven programming, which is often used when dynamically interacting with the real world. This is useful, for example, when programming graphical user interfaces, where programs will often need to react to a user clicking on the mouse or to text arriving 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, e.g., one of the above, but are a mix. Nevertheless, this book will treat each paradigm separately to emphasize its advantages and disadvantages.

1.4 Why Use F#

This book uses F#, also known as Fsharp, which is a functional-first programming language, meaning that it is designed as a functional 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. Still, it offers many advantages:

Interactive and compile mode: F# has an interactive and compile mode of operation. In interactive mode, you can write code that is executed immediately like 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 people who are not F# experts and is faster to execute.

Indentation for scope: F# uses indentation to indicate scope. Some lines of code belong together and should be executed in a certain order and may share data. 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 both via the public domain Mono platform and Microsoft's open source .Net system.

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) organized 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 IDEs such as Visual Studio (`https://www.visualstudio.com`) and Xamarin Studio (`https://www.xamarin.com`).

1.5 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 several common programming paradigms: Imperative programming mainly covered in Chapters 4 to 7, functional programming mainly covered in Chapters 8 to 10, and object-oriented programming in Chapters 15 to 17. Some general topics are given in the appendix for reference. For further reading please consult `http://fsharp.org`.

Part I

Getting started

Chapter 2

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 2.1

What is the sum of 357 and 864?

we have written the program shown in Listing 2.1. In this book, we will show many

Listing 2.1 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 2.1 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.

2.1 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 2.1 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 2.1. 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 Section 9.1.

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

Listing 2.2: 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 2.2, 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 2.2 into a file, here called `gettingStartedStump.fsx`. This file can be interpreted directly by `dotnet fsi` as shown in Listing 2.3.

Listing 2.3: Using the interpreter to execute a script.

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

Notice that in the file, "`;;`" is optional. In comparison to Listing 2.2, 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 Section 9.1, 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 2.1 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 2.1 Suffixes used, when programming F#.

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

Types are a central concept in F#. In the script 2.1 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 2.4. The interactive session displays the type using the

Listing 2.4: 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 2.1. For many problems,

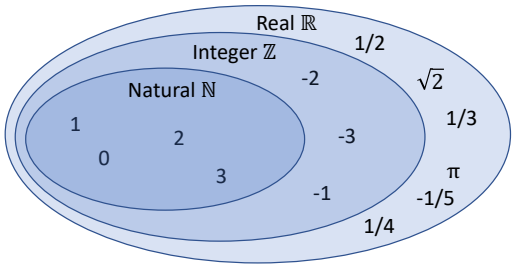


Fig. 2.1 In mathematics, the sets of natural, integer, and real numbers are each infinitely large, and real contains integers which in turn contains the set of natural numbers.

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 2.5. In this ex-

Listing 2.5: 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.

2.3 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 2.6: 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 2.1 to include it as shown in Listing 2.7. The output is the

Listing 2.7 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.

2.4 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 2.8, In this program, we find a user

Listing 2.8 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.

2.5 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 2.2

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 2.9 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"
```

2.6 Repeatedly execute code

Often code needs to be evaluated many times or looped. For example, instead of stopping the program in Listing 2.9 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 2.10. The function `readNonZeroValue` takes no input denoted by "`()`", and repeatedly

Listing 2.10 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 2.10 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 2.10 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 2.11 quickStartWhileInput.fsx:
Replacing recursion in Listing 2.10 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 2.10 and Listing 2.11, 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.

2.7 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 2.10 could look like Listing 2.12 Comments are ignored by the computer and serve solely as programmer-

Listing 2.12 quickStartRecursiveInputComments.fsx: Adding comments to Listing 2.10.

```
(*
  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.

2.8 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.

Chapter 3

Using F# as a Calculator

Abstract In the previous chapter, we introduced some key F# programming tools and

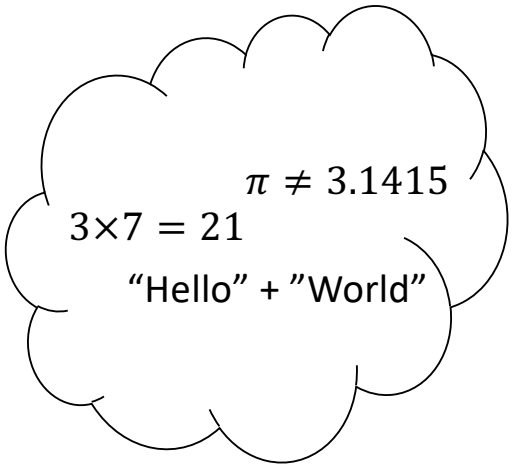


Fig. 3.1 The basis of F# are constants and expressions

concepts without going into depth on opportunities and limitations. In the following chapters, we will dive deeper into methods for solving problems by writing programs, and what facilities are available in F# to express these solutions. As a first step, we must acquaint ourselves with the basic building blocks of basic types, constants, and operators, and this chapter includes

- An introduction to the basic types and how to write constants of those types.
- Arithmetic of basic operations.

with these tools, you will be able to evaluate expressions as if F# were a simple calculator. Examples of problems, you will be able to solve after reading this chapter, is:

- What is the result of $3\cos(4 * \pi/180) + 4\sin(4 * \pi/180)$.
- Calculate how many characters are there in the text string “Hello World!”.
- What is the ASCII value of the character ’J’.
- How to convert between whole and binary numbers.

3.1 Literals and Basic Types

All programs rely on the processing of data, and an essential property of data is its *type*. A *literal* is a fixed value like the number 3, and if we type the number `3` in an interactive session at the input prompt, then F# responds as shown in Listing 3.1. What this means is that F# has inferred the type to be `int` and bound it to the

Listing 3.1: Typing the number 3.

```
> 3;;
val it: int = 3
```

identifier `it`. For more on binding and identifiers see Chapter 4. Types matter, since the operations that can be performed on integers, are quite different from those that can be performed on, e.g., strings. Therefore, the number 3 has many different representations as shown in Listing 3.2. Each literal represents the number 3, but

Listing 3.2: Many representations of the number 3 but using different types.

```

> 3;;
val it: int = 3

> 3.0;;
val it: float = 3.0

> '3';;
val it: char = '3'

> "3";;
val it: string = "3"
```

their types are different, and hence they are quite different values. The types `int` for integer numbers, `float` for floating point numbers, `bool` for Boolean values, `char` for characters, and `string` for strings of characters are the most common types of literals. A table of all *basic types* predefined in F# is given in Table 3.1. In addition to these built-in types, F# is designed such that it is easy to define new types.

Humans like to use the *decimal number* system for representing numbers. Decimal numbers are *base* 10, which means that a value is represented as two sequences of decimal digits separated by a *decimal point*, where each *digit* d has a position and a value $d \in \{0, 1, 2, \ldots, 9\}$. The part before the decimal point is called the *whole part* and the part after is called the *fractional part* of the number. An *integer* is a number with only a whole part and neither a decimal point nor a fractional part. As an example `35.7` is a decimal number, whose value is $3 \cdot 10^1 + 5 \cdot 10^0 + 7 \cdot 10^{-1}$, and `128` is an integer, whose value is $1 \cdot 10^2 + 2 \cdot 10^1 + 8 \cdot 10^0$. In F#, a decimal number is called a *floating point number*. Floating point numbers may alternatively be given

Metatype	Type name	Description
Boolean	`bool`	Boolean values true or false
Integer	`int`	Integer values from -2,147,483,648 to 2,147,483,647
	`byte`	Integer values from 0 to 255
	`sbyte`	Integer values from -128 to 127
	`int8`	Synonymous with sbyte
	`uint8`	Synonymous with byte
	`int16`	Integer values from -32768 to 32767
	`uint16`	Integer values from 0 to 65535
	`int32`	Synonymous with int
	`uint32`	Integer values from 0 to 4,294,967,295
	`int64`	Integer values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
	`uint64`	Integer values from 0 to 18,446,744,073,709,551,615
Real	`float`	64-bit IEEE 754 floating point value from $-\infty$ to ∞
	`double`	Synonymous with float
	`single`	A 32-bit floating point type
	`float32`	Synonymous with single
	`decimal`	A floating point data type that has at least 28 significant digits
Character	`char`	Unicode character
	`string`	Unicode sequence of characters
None	`unit`	The value ()
Object	`obj`	An object
Exception	`exn`	An exception

Table 3.1 List of some of the basic types. The most commonly used types are underlined. For a description of integer see Appendix B.1, for floating point numbers see Appendix B.2, for ASCII and Unicode characters see Appendix C, for objects see Chapter 15, and for exceptions see Section 12.3.

using *scientific notation*, such as `3.5e-4` and `4e2`, where the `e`-notation is translated to a value as `3.5e-4` $= 3.5 \cdot 10^{-4} = 0.00035$, and `4e2` $= 4 \cdot 10^2 = 400$.

The basic unit of information in almost all computers is the binary digit or *bit* for short. Internally, programs and data are all represented as bits, hence F# has strong support for binary numbers. A *binary number* consists of a sequence of binary digits separated by a decimal point, where each digit can have values $b \in \{0, 1\}$, and the base is 2. E.g., the binary number $101.01_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 5.25$. Subscripts are often used to indicate the base of a number, e.g., 101.01_2 and 101.01_{10} are different numbers. Since base 10 is so common, the subscript for base 10 numbers is often omitted.

Binary numbers are closely related to *octal* and *hexadecimal numbers*. Octals use 8 as their basis and hexadecimals use 16 as their basis. Each octal digit can be represented by exactly three bits, and each hexadecimal digit can be represented by exactly four bits. The hexadecimal digits use `0`–`9` to represent the values 0–9 and `a`–`f` in lower or alternatively upper case to represent the values 10-15. Thus, Octals and hexadecimals conveniently serve as shorthand for the much longer binary representation. As examples, the octal number 37_8 is $3 \cdot 8^1 + 7 \cdot 8^0 = 31$, and the hexadecimal number $f3_{16}$ is $15 \cdot 16^1 + 3 \cdot 16^0 = 243$.

To denote integers in bases different than 10, F# uses the prefix '0b' for binary, '0o' for octal, and '0x' for hexadecimal numbers. For example, the value 367_{10} may be written as an integer `367`, as a binary number `0b101101111`, as an octal number `0o557`, and as a hexadecimal number `0x16f`. In F#, the character sequences `0b12` and `ff` are not recognized as numbers.

A *character* is a *Unicode code point*, and character literals are enclosed in single quotation marks. Appendix C.3 contains more details on code points. The character type in F# is denoted *`char`*. Examples of characters are 'a', 'D', '3', and examples of non-characters are '23' and 'abc'. Some characters, such as the tabulation character, do not have a visual representation. These can still be represented as a character using *escape sequences*. A character escape sequence starts with "\" followed by letter for simple escapes such as `\t` for tabulation and `\n` for newline. Escape sequences can also be a numerical representation of a code point, and three versions exist: The trigraph `\DDD`, where `D` is a decimal digit, is used to specify the first 256 code points, the hexadecimal escape codes `\uXXXX`, where `X` is a hexadecimal digit, is used to specify the first 65536 code points, and `\UXXXXXXXX` is used to specify any of the approximately $4.3 \cdot 10^9$ possible code points. All escape sequences are shown in Table 3.2. Examples of `char` representations of the letter 'a' are: `'a'`, `'\097'`,

Character	Escape sequence	Description
BS	`\b`	Backspace
LF	`\n`	Line feed
CR	`\r`	Carriage return
HT	`\t`	Horizontal tabulation
\	`\\`	Backslash
"	`\"`	Quotation mark
'	`\'`	Apostrophe
BEL	`\a`	Bell
FF	`\f`	Form feed
VT	`\v`	Vertical tabulation
	`\uXXXX`, `\UXXXXXXXX`, `\DDD`	Unicode character ('X' is any hexadecimal digit, and 'D' is any decimal digit)

Table 3.2 Escape characters. The escape code `\DDD` is sometimes called a tricode.

`'\u0061'`, `'\U00000061'`.

A *string* is a sequence of characters enclosed in double quotation marks. Examples are `"a"`, `"this is a string"`, and `"-&#\@"`. Note that the string `"a"` and the character `'a'` are not the same. Some strings are so common that they are given special names: One or more spaces `" "` is called *whitespace*, and both `"\n"` and `"\r\n"` are called *newline*. The escape-character "\" may be used to break a line in two. This and other examples are shown in Listing 3.3. Note that the response from `fsharpi` is shown in double quotation marks, but this is not part of the string.

F# supports *literal types*, where the type of a literal is indicated as a prefix or suffix as shown in Table 3.3.

Listing 3.3: Examples of string literals.

```
> "abcde";;
val it: string = "abcde"

> "abc
  de";;
val it: string = "abc
  de"

> "abc\
  de";;
val it: string = "abcde"

> "abc\nde";;
val it: string = "abc
de"
```

Type	syntax	Examples	Value
int, int32	<int | xint> <int | xint>l	3, 0x3 3l, 0x3l	3
uint32	<int | xint>u <int | xint>ul	3u 3ul	3
byte, uint8	<int | xint>uy '<char>'B	97uy 'a'B	97
byte[]	"<string>"B @"<string>"B	"a\n"B @"a\n"B	[|97uy; 10uy|] [|97uy; 92uy; 110uy|]
sbyte, int8	<int | xint>y	3y	3
int16	<int | xint>s	3s	3
uint16	<int | xint>us	3us	3
int64	<int | xint>L	3L	3
uint64	<int | xint>UL <int | xint>uL	3UL 3uL	3
float, double	<float> <xint>LF	3.0 0x013LF	3.0 9.387247271e-323
single, float32	<float>F <float>f <xint>lf	3.0F 3.0f 0x013lf	3.0 3.0 4.4701421e-43f
decimal	<float | int>M <float | int>m	3.0M, 3M 3.0m, 3m	3.0
string	"<string>" @"<string>" """<string>"""	"\"quote\".\n" @"""quote"".\n" """"quote".\n"""	"quote".<newline> "quote".\n. "quote".\n

Table 3.3 List of literal types. The syntax notation <> means that the programmer replaces the brackets and content with a value of the appropriate form. The <xint> is one of the integers on hexadecimal, octal, or binary forms such as 0x17, 0o21, and 0b10001. The [| |] brackets means that the value is an array, see Section 13.4 for details.

The literal type is closely connected to how the values are represented internally. For example, a value of type `int32` uses 32 bits and can be both positive and negative, while a `uint32` value also uses 32 bits, but is unsigned. A `byte` is an 8-bit number, and `sbyte` is a signed 8-bit number. Values of type `float` use 64 bits, while `float32` only uses 32 bits. The number of bits used to represent numbers directly relates to the

range and precision these types can represent. This is summarized in Table 3.1 and discussed in more detail in Appendix B. String literals may be *verbatim* by the @-notation or triple double quotation marks, meaning that the escape sequences are not converted to their code point. The two types of string verbatim treat quotation marks differently, as illustrated in the table. Further examples are shown in Listing 3.4.

Listing 3.4: Named and implied literals.

```
> 3;;
val it: int = 3

> 4u;;
val it: uint32 = 4u

> 5.6;;
val it: float = 5.6

> 7.9f;;
val it: float32 = 7.900000095f

> 'A';;
val it: char = 'A'

> 'B'B;;
val it: byte = 66uy

> "ABC";;
val it: string = "ABC"

> @"abc\nde";;
val it: string = "abc\nde"
```

Many basic types are compatible, and the type of a literal may be changed by *typecasting*. An example of casting to a `float` is shown in Listing 3.5. When `float`

Listing 3.5: Casting an integer to a floating point number.

```
> float 3;;
val it: float = 3.0
```

is given an argument, then it acts as a function rather than a type, and for the integer `3` it returns the floating point number `3.0`. For more on functions see Chapter 4. Boolean values are often treated as integer values 0 and 1, but no short-hand function names exist for their conversions. Instead, use functions from the `System.Convert` family of functions, as demonstrated in Listing 3.6. Here `System.Convert.ToBoolean` is the identifier of a function `ToBoolean`, which is a *member* of the *class* `Convert` that is included in the *namespace* `System`. Namespaces, classes, and members will be discussed in Chapter 9.

Typecasting is often a destructive operation, e.g., typecasting a `float` to `int` removes the fractional part without rounding as shown in Listing 3.7. Here we typecasted to

Listing 3.6: Casting booleans.

```
> System.Convert.ToBoolean 1;;
val it: bool = true

> System.Convert.ToBoolean 0;;
val it: bool = false

> System.Convert.ToInt32 true;;
val it: int = 1

> System.Convert.ToInt32 false;;
val it: int = 0
```

Listing 3.7: Fractional part is removed by downcasting.

```
> int 357.6;;
val it: int = 357
```

a lesser type, in the sense that the set of integers is a subset of floating point numbers, and this is called *downcasting*. The opposite is called *upcasting* and is often non-destructive, as Listing 3.5 showed. Since floating point numbers are a superset of integers, the value is retained. As a side note, *rounding* a number $y.x$, where y is the *whole part* and x is the *fractional part*, is the operation of mapping numbers in the interval $y.x \in [y.0, y.5)$ to y, and those in $y.x \in [y.5, y+1)$ to $y+1$. This can be performed by downcasting, as shown in Listing 3.8. I.e., $357.6 + 0.5 = 358.1$

Listing 3.8: Rounding by modified downcasting.

```
> int (357.6 + 0.5);;
val it: int = 358
```

and removing the fractional part by downcasting results in 358, which is the correct answer.

3.2 Operators on Basic Types

Expressions are the basic building block of all F# programs, and this section will discuss operator expressions on basic types. A typical calculation, such used in Listing 3.8, is

$$\underbrace{357.6}_{\text{operand}} \quad \underbrace{+}_{\text{operator}} \quad \underbrace{0.5}_{\text{operand}} \tag{3.1}$$

is an example of an arithmetic *expression*, and the above expression consists of two *operands* and an *operator*. Since this operator takes two operands, it is called a

binary operator. The expression is written using *infix notation*, since the operands appear on each side of the operator.

In order to discuss general programming structures, we will use simplified language to describe valid syntactical structures. In this simplified language, the syntax of basic binary operators is shown in the following.

Listing 3.9: Syntax for a binary expression.

```
1 <expr><op><expr>
```

Here <expr> is any expression supplied by the programmer, and <op> is a binary infix operator. F# supports a range of arithmetic binary infix operators on its built-in types, such as addition, subtraction, multiplication, division, and exponentiation, using the "+", "-", "*", "/", "**" lexemes, respectively. Not all operators are defined for all types, e.g., addition is defined for integer and float types as well as for characters and strings, but multiplication is only defined for integer and floating-point types. A complete list of built-in operators on basic types is shown in Table 3.4 and 3.5, and a range of mathematical functions is shown in Table 3.6. Note that

Operator	bool	ints	floats	char	string	Example	Result	Description
+		✓	✓	✓	✓	`5 + 2`	`7`	Addition
-		✓	✓			`5.0 - 2.0`	`3.0`	Subtraction
*		✓	✓			`5 * 2`	`10`	Multiplication
/		✓	✓			`5.0 / 2.0`	`2.5`	Division
%		✓	✓			`5 % 2`	`1`	Remainder
**			✓			`5.0 ** 2.0`	`25.0`	Exponentiation
+		✓	✓			`+3`	`3`	identity
-		✓	✓			`-3.0`	`-3.0`	negation
&&	✓					`true && false`	`false`	boolean and
||	✓					`true || false`	`true`	boolean or
not	✓					`not true`	`false`	boolean negation
&&&		✓				`0b101 &&& 0b110`	`0b100`	bitwise boolean and
|||		✓				`0b101 ||| 0b110`	`0b111`	bitwise boolean or
^^^		✓				`0b101 ^^^ 0b110`	`0b011`	bitwise boolean exclusive or
<<<		✓				`0b110uy <<< 2`	`0b11000uy`	bitwise left shift
>>>		✓				`0b110uy >>> 2`	`0b1uy`	bitwise right shift
~~		✓				`~~~0b110uy`	`0b11111001uy`	bitwise boolean negation

Table 3.4 Arithmetic operators on basic types. Ints and floats means all built-in integer and float types. Note that for the bitwise operations, digits `0` and `1` are taken to be `true` and `false`.

expressions can themselves be arguments to expressions, and thus, `4+5+6` is also a legal statement. Technically, F# interprets the expression as `(4+5)+6` meaning that first `4+5` is evaluated according the <expr><op><expr> syntax. Then the result replaces the parenthesis to yield `9+6`, which, once again, is evaluated according to the <expr><op><expr> syntax to give `15`. This is called *recursion*, which is the name for a type of rule or function that uses itself in its definition. See Chapter 8 for more on recursive functions.

Operator	bool	ints	floats	char	string	Example	Result	Description
<	✓	✓	✓	✓	✓	`true < false`	`false`	Less than
>	✓	✓	✓	✓	✓	`5 > 2`	`true`	Greater than
=	✓	✓	✓	✓	✓	`5.0 = 2.0`	`false`	Equal
<=	✓	✓	✓	✓	✓	`'a' <= 'b'`	`true`	Less than or equal
>=	✓	✓	✓	✓	✓	`"ab" >= "cd"`	`false`	Greater than or equal
<>	✓	✓	✓	✓	✓	`5 <> 2`	`true`	Not equal

Table 3.5 Comparison operators on basic types. Types cannot be mixed, e.g., `3<'a'` is a syntax error.

Operator	bool	ints	floats	char	string	Example	Result	Description
`abs`		✓	✓			`abs -3`	`3`	Absolute value
`acos`			✓			`acos 0.8`	`0.644`	Inverse cosine
`asin`			✓			`asin 0.8`	`0.927`	Inverse sinus
`atan`			✓			`atan 0.8`	`0.675`	Inverse tangent
`atan2`			✓			`atan2 0.8 2.3`	`0.335`	Inverse tangentvariant
`ceil`			✓			`ceil 0.8`	`1.0`	Ceiling
`cos`			✓			`cos 0.8`	`0.697`	Cosine
`exp`			✓			`exp 0.8`	`2.23`	Natural exponent
`floor`			✓			`floor 0.8`	`0.0`	Floor
`log`			✓			`log 0.8`	`-0.223`	Natural logarithm
`log10`			✓			`log10 0.8`	`-0.0969`	Base-10 logarithm
`max`		✓	✓	✓	✓	`max 3.0 4.0`	`4.0`	Maximum
`min`		✓	✓	✓	✓	`min 3.0 4.0`	`3.0`	Minimum
`pown`		✓				`pown 3 2`	`9`	Integer exponent
`round`			✓			`round 0.8`	`1.0`	Rounding
`sign`		✓	✓			`sign -3`	`-1`	Sign
`sin`			✓			`sin 0.8`	`0.717`	Sinus
`sqrt`			✓			`sqrt 0.8`	`0.894`	Square root
`tan`			✓			`tan 0.8`	`1.03`	Tangent

Table 3.6 Predefined functions for arithmetic operations.

Unary operators take only one argument and have the syntax:

Listing 3.10: A unary expressions.

```
1 <op><expr>
```

An example of a unary operator is `-3`, where `-` here is used to negate a positive integer. Since the operator appears before the operand, it is a *prefix operator*.

The concept of *precedence* is an important concept in arithmetic expressions. If parentheses are omitted in Listing 3.8, then F# will interpret the expression as `(int 357.6) + 0.5`, which is erroneous since the addition of an integer with a float is undefined. This is an example of precedence, i.e., function evaluation takes precedence over addition which means that function evaluation is performed first and

addition second. Consider the arithmetic expression shown in Listing 3.11. Here, the

Listing 3.11: A simple arithmetic expression.

```
> 3 + 4 * 5;;
val it: int = 23
```

addition and multiplication functions are shown in infix notation with the *operator* lexemes "+" and "*". To arrive at the resulting value 23, F# has to decide in which order to perform the calculation. There are 2 possible orders, `3 + (4 * 5)` and `(3 + 4) * 5` that gives different results. For integer arithmetic, the correct order is, of course, multiplication before addition, and we say that multiplication takes *precedence* over addition. Every atomic operation that F# can perform is ordered in terms of its precedence, and for some common built-in operators shown in Table 3.7, the precedence is shown by the order they are given in the table.

Operator	Associativity	Description

`+<expr>`
`-<expr>`
`~~~<expr>`
	Left	Unary identity, negation, and bitwise negation operators

`f <expr>`	Left	Function application
`<expr> ** <expr>`	Right	Exponentiation

`<expr> * <expr>`
`<expr> / <expr>`
`<expr> % <expr>`
	Left	Multiplication, division and remainder

`<expr> + <expr>` `<expr> - <expr>`	Left	Addition and subtraction binary operators
`<expr> ^^^ <expr>`	Right	Bitwise exclusive or

`<expr> < <expr>`
`<expr> <= <expr>`
`<expr> > <expr>`
`<expr> >= <expr>`
`<expr> = <expr>`
`<expr> <> <expr>`
`<expr> <<< <expr>`
`<expr> >>> <expr>`
`<expr> &&& <expr>`
`<expr> ||| <expr>`
	Left	Comparison operators, bitwise shift, and bitwise 'and' and 'or'.

`<expr> && <expr>`	Left	Boolean and
`<expr> || <expr>`	Left	Boolean or

Table 3.7 Some common operators, their precedence, and their associativity. Rows are ordered from highest to lowest precedences, such that `<*expr*> * <*expr*>` has higher precedence than `<*expr*> + <*expr*>`. Operators in the same row have the same precedence..

Associativity describes the order in which calculations are performed for binary operators of the same precedence. Some operator's associativity are given in Table 3.7. In the table we see that "*" is left associative, which means that `3.0 * 4.0 * 5.0` is evaluated as `(3.0 * 4.0) * 5.0`. Conversely, ** is right-associative, so `4.0**3.0**2.0` is evaluated as `4.0**(3.0**2.0)`. For some operators, like multiplication, association matters little, e.g., $4 * 3 * 2 = 4 * (3 * 2) = (4 * 3) * 2$, and for other operators, like exponentiation, the association makes a huge difference, e.g.,

★ $4^{(3^2)} \neq (4^3)^2$. Examples of this are shown in Listing 3.12. **Whenever in doubt of**

Listing 3.12: Precedence rules define implicit parentheses.

```
> 4.0 * 3.0 * 2.0;;
val it: float = 24.0

> (4.0 * 3.0) * 2.0;;
val it: float = 24.0

> 4.0 * (3.0 * 2.0);;
val it: float = 24.0

> 4.0 ** 3.0 ** 2.0;;
val it: float = 262144.0

> (4.0 ** 3.0) ** 2.0;;
val it: float = 4096.0

> 4.0 ** (3.0 ** 2.0);;
val it: float = 262144.0
```

association or any other basic semantic rules, it is a good idea to use parentheses. It is also a good idea to test your understanding of the syntax and semantic rules by making a simple script.

3.3 Boolean Arithmetic

Boolean arithmetic is the basis of almost all computers and is particularly important for controlling program flow, which will be discussed in Section 13.2. Boolean values are one of 2 possible values, true or false, which is also sometimes written as 1 and 0. Basic operations on Boolean values are '*and*', '*or*', and '*not*', which in F# are written respectively as the binary operators `&&`, `||`, and the function `not`. Since the domain of Boolean values is so small, all possible combinations of input on these values can be written on the tabular form, known as a *truth table*, and the truth tables for the basic Boolean operators and functions are shown in Table 3.8. A good mnemonic for remembering the result of the 'and' and 'or' operators is to

a	b	a && b	a || b	not a
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Table 3.8 Truth table for boolean 'and', 'or', and 'not' operators. Value 0 is false and 1 is true.

use 1 for true, 0 for false, multiplication for the Boolean 'and' operator, and addition for the Boolean 'or' operator, e.g., true and false in this mnemonic translates to

$1 \cdot 0 = 0$, and the result translates back to the Boolean value false. In F#, the truth table for the basic Boolean operators can be produced by a program, as shown in Listing 3.13. Here, we used the `printfn` function to present the results of many

Listing 3.13: Boolean operators and truth tables.

```
1
2  > printfn "a b a*b a+b not a"
3  printfn "%A %A %A %A %A"
4    false false (false && false) (false || false) (not false)
5  printfn "%A %A %A %A %A"
6    false true (false && true) (false || true) (not false)
7  printfn "%A %A %A %A %A"
8    true false (true && false) (true || false) (not true)
9  printfn "%A %A %A %A %A"
10   true true (true && true) (true || true) (not true);;
11 a b a*b a+b not a
12 false false false false true
13 false true false true true
14 true false false true false
15 true true true true false
16 val it: unit = ()
```

expressions on something that resembles a tabular form. The spacing produced using the `printfn` function is not elegant, and in Chapter 12 we will discuss better options for producing more beautiful output. Notice that the arguments for `printfn` were given on the next line with indentation. The indentation is an important part of telling F# which part of what you write belongs together. This is an example of the so-called lightweight syntax. Generally, F# ignores newlines and whitespaces except when using the lightweight syntax. The difference between verbose and lightweight syntax is discussed in Chapter 4.

3.4 Integer Arithmetic

The set of integers is infinitely large, but since all computers have limited resources, it is not possible to represent it in its entirety. The various integer types listed in Table 3.1 are finite subsets reduced by limiting their ranges. An in-depth description of integer implementation can be found in Appendix B. The type `int` is the most common type.

Table 3.4–3.6 give examples of operators and functions pre-defined for integer types. Notice that fewer functions are available for integers than for floating point numbers. For most addition, subtraction, multiplication, and negation, the result is straightforward. However, performing arithmetic operations on integers requires extra care, since the result may cause *overflow* and *underflow*. For example, an `sbyte` is speci-

fied using the "y"-literal and can hold values $[-128 \ldots 127]$. This causes problems in the example in Listing 3.14. Here $100+30 = 130$, which is larger than the biggest

Listing 3.14: Adding integers may cause overflow.

```
> 100y;;
val it: sbyte = 100y

> 30y;;
val it: sbyte = 30y

> 100y + 30y;;
val it: sbyte = -126y
```

`sbyte`, and the result is an overflow. Similarly, we get an underflow, when the arithmetic result falls below the smallest value storable in an `sbyte`, as demonstrated in Listing 3.15. I.e., we were expecting a negative number but got a positive number

Listing 3.15: Subtracting integers may cause underflow.

```
> -100y - 30y;;
val it: sbyte = 126y
```

instead.

The overflow error in Listing 3.14 can be understood in terms of the binary representation of integers: In binary, $130 = 10000010_2$, and this binary pattern is interpreted differently as `byte` and `sbyte`, see Listing 3.16. That is, for signed bytes, the left-

Listing 3.16: The leftmost bit is interpreted differently for signed and unsigned integers, which gives rise to potential overflow errors.

```
> 0b10000010uy;;
val it: byte = 130uy

> 0b10000010y;;
val it: sbyte = -126y
```

most bit is used to represent the sign, and since the addition of $100 = 01100100_2$ and $30 = 00011110_b$ is $130 = 10000010_2$, which causes the left-most bit to be used, this is wrongly interpreted as a negative number when stored in an `sbyte`. Similar arguments can be made explaining underflows.

The operator discards the fractional part after division, and the *integer remainder* operator calculates the remainder after integer division, as demonstrated in Listing 3.17. Together, the integer division and remainder can form a lossless representation of the original number, see Listing 3.18. Here we see that integer division of 7 by 3 followed by multiplication by 3 is less than 7, and that the difference is `7 % 3`.

Notice that neither overflow nor underflow error gave rise to an error message, which is why such bugs are difficult to find. Dividing any non-zero number by 0 is infinite,

Listing 3.17: Integer division and remainder operators.

```
1
2 > 7 / 3;;
3 val it: int = 2
4
5 > 7 % 3;;
6 val it: int = 1
```

Listing 3.18: Integer division and remainder is a lossless representation of an integer, compare with Listing 3.17.

```
1 > (7 / 3) * 3;;
2 val it: int = 6
3
4 > (7 / 3) * 3 + (7 % 3);;
5 val it: int = 7
```

which is also outside the domain of any of the integer types, but in this case, F# casts an *exception*, as shown in Listing 3.19. The output looks daunting at first sight,

Listing 3.19: Integer division by zero causes an exception runtime error.

```
1 > 3/0;;
2 System.DivideByZeroException: Attempted to divide by zero.
3    at <StartupCode$FSI_0002>.$FSI_0002.main@()
```

but the first and last lines of the error message are the most important parts, which tell us what exception was cast and why the program stopped. The middle contains technical details concerning which part of the program caused the error and can be ignored for the time being. Exceptions are a type of *runtime error*, and are discussed in Section 12.3

Integer exponentiation is not defined as an operator but is available as the built-in function `pown`. This function is demonstrated in Listing 3.20 for calculating 2^5.

Listing 3.20: Integer exponent function.

```
1 > pown 2 5;;
2 val it: int = 32
```

For binary arithmetic on integers, the following operators are available: `<leftExpr> <<< <rightExpr>`, which shifts the bit pattern of `<leftExpr>` `<rightExpr>` positions to the left while inserting 0's to right; `<leftExpr> >>> <rightExpr>`, which shifts the bit pattern of `<leftExpr>` `<rightExpr>` positions to the right while inserting 0's to left; `~~~ <expr>` returns a new integer, where all 0 bits are changed to 1 bits and vice-versa; `<expr> &&& <expr>` returns the result of taking the Boolean 'and' operator position-wise; `<expr> ||| <expr>` returns the result of taking the Boolean 'or' operator position-wise; and `<expr> ^^^ <expr>` returns the result of the Boolean 'xor' operator defined by the truth table in Table 3.9.

a	b	a ^^^ b
false	false	false
false	true	true
true	false	true
true	true	false

Table 3.9 Boolean exclusive or truth table.

3.5 Floating Point Arithmetic

Like integers, the set of reals is also infinitely large, hence, floating point types are finite subsets reduced by sampling the space of reals. An in-depth description of floating point implementations can be found in Appendix B. The type `float` is the most common type.

Table 3.4–3.6 give examples of operators and functions pre-defined for floating point types. Note that the remainder operator for floats calculates the remainder after division and discards the fractional part, see Listing 3.21. The remainder for

Listing 3.21: Floating point division and remainder operators.

```
> 7.0 / 2.5;;
val it: float = 2.8

> 7.0 % 2.5;;
val it: float = 2.0
```

floating point numbers can be fractional, but division, discarding fractional part, and the remainder is still a lossless representation of the original number, as demonstrated in Listing 3.22.

Listing 3.22: Floating point division, downcasting, and remainder is a lossless representation of a number.

```
> float (int (7.0 / 2.5));;
val it: float = 2.0

> (float (int (7.0 / 2.5))) * 2.5;;
val it: float = 5.0

> (float (int (7.0 / 2.5))) * 2.5 + 7.0 % 2.5;;
val it: float = 7.0
```

Arithmetic using `float` will not cause over- and underflow problems, since the IEEE 754 standard includes the special numbers $\pm\infty$ and NaN. As shown in Listing 3.23, no exception is thrown. However, the `float` type has limited precision since there is only a finite number of numbers that can be stored in a float. E.g., addition and subtraction can give surprising results, as demonstrated in Listing 3.24. That is, addition and subtraction associates to the left, hence the expression is interpreted as `(357.8 + 0.1) - 357.9` and we see that we do not get the expected 0. The

Listing 3.23: Floating point numbers include infinity and Not-a-Number.

```
> 1.0/0.0;;
val it: float = infinity

> 0.0/0.0;;
val it: float = nan
```

Listing 3.24: Floating point arithmetic has finite precision.

```
> 357.8 + 0.1 - 357.9;;
val it: float = 5.684341886e-14
```

reason is that the calculation is done stepwise, and in the process, the numbers are represented using the imprecise floating point standard. Thus,`357.8 + 0.1` is represented as a number close to but not identical to what `357.9` is represented as, and thus, when subtracting these two representations, we get a very small nonzero number. Such errors tend to accumulate, and comparing the result of expressions of floating point values should, therefore, be treated with care. Thus, **equivalence of two floating point expressions should only be considered up to sufficient precision, e.g., comparing `357.8 + 0.1` and `357.9` up to `1e-10` precision should be tested as, `abs ((357.8 + 0.1) - 357.9) < 1e-10`.** ⋆

3.6 Char and String Arithmetic

Addition is the only operator defined for characters. Nevertheless, character arithmetic is often done by casting to an integer. A typical example is the conversion of character case, e.g., to convert the lowercase character 'z' to uppercase. Here, we use the *ASCIIbetical order*, add the difference between any Basic Latin Block letters in upper- and lowercase as `integers`, and cast back to `char`, see Listing 3.25. I.e., the code point difference between the upper and lower case for any alphabetical

Listing 3.25: Converting case by casting and integer arithmetic.

```
> char (int 'z' - int 'a' + int 'A');;
val it: char = 'Z'
```

character 'a' to 'z' is constant, hence we can change the case by adding or subtracting the difference between any corresponding character. Unfortunately, this does not generalize to characters from other languages.

A large collection of operators and functions exist for `string`. The simplest is concatenation using the "+" operator, as demonstrated in Listing 3.26. Characters and strings cannot be concatenated, which is why the above example used the string of a space `" "` instead of the space character `' '`. The characters of a string may

Listing 3.26: Example of string concatenation.

```
> "hello" + " " + "world";;
val it: string = "hello world"
```

be indexed as using the `[]` notation. This is demonstrated in Listing 3.27. No-

Listing 3.27: String indexing using square brackets.

```

> "abcdefg"[0];;
val it: char = 'a'

> "abcdefg"[3];;
val it: char = 'd'

> "abcdefg"[3..];;
val it: string = "defg"

> "abcdefg"[..3];;
val it: string = "abcd"

> "abcdefg"[1..3];;
val it: string = "bcd"

> "abcdefg"[*];;
val it: string = "abcdefg"
```

tice that the first character has index 0, and to get the last character in a string, we use the string's *`Length` property*. A Property is an extra piece of information associated with a given value. This is done as shown in Listing 3.28. Since

Listing 3.28: String Length property and string indexing.

```
> "abcdefg".Length;;
val it: int = 7

> "abcdefg"[7-1];;
val it: char = 'g'
```

index counting starts at 0, and since the string length is 7, the index of the last character is 6. There is a long list of built-in functions in `System.String` for working with strings, see `https://fsharp.github.io/fsharp-core-docs/reference/fsharp-core-stringmodule.html` for further details.

The *dot notation* is an example of Structured programming, where technically speaking, the string `"abcdefg"` is an immutable *object* of *class* `string`, `[]` is an object *method*, and `Length` is a property. For more on objects, classes, and methods, see Chapter 15.

Strings are compared letter by letter. For two strings to be equal, they must have the same length and all the letters must be identical. E.g., `"abs" = "absalon"` is false,

while `"abs" = "abs"` is true. The "<>" operator is the boolean negation of the "=" operator, e.g., `"abs" <> "absalon"` is true, while `"abs" <> "abs"` is false. For the "<" , "<=", ">", and ">=" operators, the strings are ordered alphabetically, such that `"abs" < "absalon" && "absalon" < "milk"` is true, that is, the "<" operator on two strings is true if the left operand should come before the right when sorting alphabetically. The algorithm for deciding the boolean value of `leftOp < rightOp` is as follows: we start by examining the first character, and if `leftOp[0]` and `rightOp[0]` are different, then `leftOp < rightOp` is equal to `leftOp[0] < rightOp[0]`. E.g., `"milk" < "abs"` is the same as `'m' < 'a'`, which is false, since the letter 'm' does not come before the letter 'a' in the alphabet, or more precisely, the codepoint of 'm' is not less than the codepoint of 'a'. If `leftOp[0]` and `rightOp[0]` are equal, then we move on to the next letter and repeat the investigation, e.g., `"abe" < "abs"` is true, since `"ab" = "ab"` is true and `'e' < 's'` is true. If we reach the end of either of the two strings, then the shorter word is smaller than the longer word, e.g., `"abs" < "absalon"` is true, while `"abs" < "abs"` is false. The "<=", ">", and ">=" operators are defined in a similar manner.

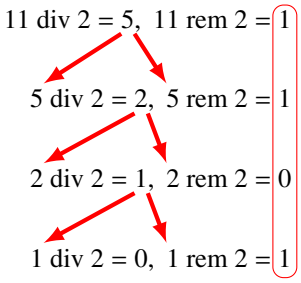
3.7 Programming Intermezzo: Hand Conversion Between Decimal and Binary Numbers

Conversion of integers between decimal and binary form is a key concept one must grasp in order to understand some of the basic properties of calculations on the computer. Converting from binary to decimal is straightforward if using the power-of-two algorithm, i.e., given a sequence of $n + 1$ binary digits b_i written as $b_n b_{n-1} \ldots b_0$, and where b_n and b_0 are the most and least significant bits respectively, then the decimal value is calculated as,

$$v = \sum_{i=0}^{n} b_i 2^i \tag{3.2}$$

For example, $10011_2 = 1 + 2 + 16 = 19$. Converting from decimal to binary is a little more complex, but a simple divide-by-two algorithm exists. The key to understanding the divide-by-two algorithm is to realize that dividing a number by two is equivalent to shifting its binary representation from one position to the right. E.g., $10 = 1010_2$ and $10/2 = 5 = 101_2$. Odd numbers have $b_0 = 1$, e.g., $11_{10} = 1011_2$ and $11_{10}/2 = 5.5 = 101.1_2$. Hence, if we divide any number by two and get a non-integer number, then its least significant bit was 1. Another way to express this is to say that the least significant bit is the remainder after integer division by two. Sequential application of this idea leads directly to the divide-by-two algorithm. E.g., if we were to convert the number 11_{10} in decimal form to binary form, we

would perform the following steps:



Here we used div and rem to signify the integer division and remainder operators. The algorithm stops when the result of integer division is zero. Reading off the remainder from below and up, we find the sequence 1011_2, which is the binary form of the decimal number 11_{10}. Using the interactive mode, we can perform the same calculation, as shown in Listing 3.29.

Listing 3.29: Converting the number 11_{10} to binary form.

```
> printfn "(%d, %d)" (11 / 2) (11 % 2);;
(5, 1)
val it : unit = ()
> printfn "(%d, %d)" (5 / 2) (5 % 2);;
(2, 1)
val it : unit = ()
> printfn "(%d, %d)" (2 / 2) (2 % 2);;
(1, 0)
val it : unit = ()
> printfn "(%d, %d)" (1 / 2) (1 % 2);;
(0, 1)
val it : unit = ()
```

Thus, by reading the second integer-response from `printfn` from below and up, we again obtain the binary form of 11_{10} to be 1011_2. For integers with a fractional part, the divide-by-two algorithm may be used on the whole part, while multiply-by-two may be used in a similar manner on the fractional part.

3.8 Key Concepts and Terms in This Chapter

In this chapter you have learned about:

- the **basic types**: **int** of various kinds which are all a subset of integers, **float** of various kinds which are all subsets of reals, **bool** which captures the notion of true and false, and **char** and **string** which holds characters and sequences of characters;
- how to write constants of the basic types, which are called **literals**;
- **operators** such as "+" and "-" and whose arguments are called **operands**;
- **unary** and **binary** operators;
- **escape sequences** for characters and strings;
- how to get the **length** of a string using the **dot** notation, and how to extract substrings using the **slice** notation.

Chapter 4

Values, Functions, and Statements

Abstract In the previous chapter, you got an introduction to the fundamental concepts of booleans, numbers, characters, and strings, how to perform calculations with them, and some of the limitations they have on the computer. This allows us to perform simple calculations as if the computer was an advanced pocket calculator. In this chapter, we will look at how we can:

- Make assign values to names to make programs that are easier to read and write.
- Organise lines of code in functions, to make the same line reusable, and to make programs shorter, and easier to understand.
- Use operators as functions.
- Conditionally execute code.
- How we can simulate the computer's execution of code by tracing-by-hand, to improve our understanding of, how it interprets the code, we write, and to find errors.

Examples of problems, you can solve, after having read this chapter are:

- Write a function that given the parameters a and b in $f(x) = ax + b$, find the value of x when $f(x) = 0$.
-

In this chapter, we will see how we can bind expressions to identifiers either as new constants, functions, or operators, how this saves time when building large programs, and how this makes programs easier to read and debug. As an example, consider the following problem,

Problem 4.1

or given set constants a, b, and c, solve for x in

$$ax^2 + bx + c = 0 \tag{4.1}$$

To solve for x we use the quadratic formula from elementary algebra,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \tag{4.2}$$

which gives the general solution for any values of the coefficients. Here, we will assume a positive discriminant, $b^2 - 4ac > 0$. In order to write a program where the code may be reused later, we define a function

```
discriminant : float -> float -> float -> float
```

that is, a function that takes 3 arguments, `a`, `b`, and `c`, and calculates the discriminant. Likewise, we will define

```
positiveSolution : float -> float -> float -> float
```

and

```
negativeSolution : float -> float -> float -> float
```

that also take the polynomial's coefficients as arguments and calculate the solution corresponding to choosing the positive and negative sign for ± in the equation. Details on function definition is given in Section 4.2. Our solution thus looks like Listing 4.1. Here, we have further defined names of values `a`, `b`, and `c` which are used as inputs to our functions, and the results of function application are bound to the names `d`, `xn`, and `xp`. The names of functions and values given here are examples of identifiers, and with these, we may reuse the quadratic formulas and calculated values later, while avoiding possible typing mistakes and reducing the amount of code that needs to be debugged.

The use of identifiers is central in programming. For F#, not to be confused with built-in functionality, identifiers must follow a specific set of rules:

Identifier

Listing 4.1 identifiersExample.fsx:
Finding roots for quadratic equations using function name binding.

```
let discriminant a b c = b ** 2.0 - 4.0 * a * c
let positiveSolution a b c = (-b + sqrt (discriminant a b c))
   / (2.0 * a)
let negativeSolution a b c = (-b - sqrt (discriminant a b c))
   / (2.0 * a)

let a = 1.0
let b = 0.0
let c = -1.0
let d = discriminant a b c
let xp = positiveSolution a b c
let xn = negativeSolution a b c
do printfn "0 = %A * x ** 2.0 + %A * x + %A" a b c
do printfn "  has discriminant %A and solutions %A and %A" d
   xn xp
```

```
$ dotnet fsi identifiersExample.fsx
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has discriminant 4.0 and solutions -1.0 and 1.0
```

- Identifiers are used as names for values, functions, types etc.
- They must start with a Unicode letter or underscore '_', but can be followed by zero or more letters, digits, and a range of special characters except for SP, LF, and CR (space, line feed, and carriage return). See Appendix C.3 for more on codepoints that represent letters.
- They can also be a sequence of identifiers separated by a period.
- They cannot be keywords, see Table 4.1.

Examples of identifiers are: `a`, `theCharacter9`, `Next_Word`, `_tok`, and `f.sharp.rocks`. Since programmers often work in a multilingual environment dominated by the English language it is advisable to **restrict identifiers to use letters from the English alphabet, numbers, periods, and '_'.** However, the number of possible identifiers is enormous. The full definition refers to the Unicode general categories described in Appendix C.3, and there are currently 19.345 possible Unicode code points in the latter category and 2.245 possible Unicode code points in the special character category. ⋆

Identifiers may be used to carry information about their intended content and use, and a careful selection of identifiers can aid programmers to communicate thoughts about the code. Thus, identifiers are often a word or several concatenated words conveying some relevant meaning. For example, in the function definition `let discriminant a b c = b ** 2.0 - 4.0 * a * c`, the function identifier has

Type	Keyword
Regular	`abstract`, `and`, `as`, `assert`, `base`, `begin`, `class`, `default`, `delegate`, `do`, `done`, `downcast`, `downto`, `elif`, `else`, `end`, `exception`, `extern`, `false`, `finally`, `for`, `fun`, `function`, `global`, `if`, `in`, `inherit`, `inline`, `interface`, `internal`, `lazy`, `let`, `match`, `member`, `module`, `mutable`, `namespace`, `new`, `null`, `of`, `open`, `or`, `override`, `private`, `public`, `rec`, `return`, `sig`, `static`, `struct`, `then`, `to`, `true`, `try`, `type`, `upcast`, `use`, `val`, `void`, `when`, `while`, `with`, and `yield`.
Reserved	`atomic`, `break`, `checked`, `component`, `const`, `constraint`, `constructor`, `continue`, `eager`, `fixed`, `fori`, `functor`, `include`, `measure`, `method`, `mixin`, `object`, `parallel`, `params`, `process`, `protected`, `pure`, `recursive`, `sealed`, `tailcall`, `trait`, `virtual`, and `volatile`.
Symbolic	`let!`, `use!`, `do!`, `yield!`, `return!`, `|`, `->`, `<-`, `.`, `:`, `(`, `)`, `[`, `]`, `[<`, `>]`, `[|`, `|]`, `{`, `}`, `'`, `#`, `:?>`, `:?`, `:>`, `..`, `::`, `:=`, `;;`, `;`, `=`, `_`, `?`, `??`, `(*)`, `<@`, `@>`, `<@@`, and `@@>`.
Reserved symbolic	~ and `

Table 4.1 Table of (possibly future) *keywords* and symbolic keywords in F#.

been chosen to be `discriminant`. F# places no special significance on the word 'discriminant', and the program would work exactly the same had the function been called `let f a b c = b ** 2.0 - 4.0 * a * c`. However, to programmers, the word 'discriminant' informs us of the intended role of the function and thus
★ is much preferred. This is a general principle: **identifier names should be chosen to reflect their semantic value.**. The arguments `a`, `b`, and `c` are short, but adheres to a textbook tradition of elementary algebra. Again, we might as well have used, `let discriminant c a b = a ** 2.0 - 4.0 * c * b`, which is semantically identical to the original expression, but due to tradition, this would con-
★ fuse most readers of the code. Thus, **identifier names should be chosen consistently with the readers' traditions.** Finally, identifiers are often concatenations of words, as `positiveSolution` in Listing 4.1. Concatenations can be difficult to read. Without the capitalization of the second word, we would have had `positivesolution`. This is readable at most times but takes longer time to understand in general. Typical solutions are to use a separator, such as `positive_solution`, *lower camel case* also known as *mixed case* as in the example `positiveSolution`, and *upper camel case* also known as *pascal case* as `PositiveSolution`. In this book, we use the lower camel case except where F# requires a capital first letter. Again, the choice does not influence what a program does, only how readable it is to a fellow programmer.
★ The important part is that **identifier names consisting of concatenated words are often preferred over names with few characters, and concatenation should be emphasized, e.g., by camel casing.** Choosing the length of identifier names is a balancing act, since when working with large programs, very long identifier names can be tiresome to write, and a common practice is that the length of identifier names is proportional to the complexity of the program. I.e., complex programs use long names, simple programs use short names. What is complex and what is simple is naturally in the eye of the beholder, but when you program, remember that a future reader of the program most likely has not had time to work with the problem as long

as the programmer, thus **choose identifier names as if you were to explain the meaning of a program to a knowledgeable outsider.** ⋆

Another key concept in F# is expressions. An expression can be a mathematical expression, such as $3 * 5$, a function application, such as $f3$, and many other things. Central in this chapter is the binding of values and functions to identifiers, which is done with the keyword `let`, e.g., `let a = 1.0`.

Expressions are the main workhorse of F# and have an enormous variety in how they may be written. We will in this book gradually work through some of the more important facets.

Expressions

- An Expression is a computation such as `3 * 5`.
- They can be value bindings between identifiers and expressions that evaluate to a value or a function, see Sections 4.1 and 4.2.
- They can be `do`-bindings that produce side-effects and whose results are ignored, see Section 4.2
- They can be assignments to variables, see Section 4.1.
- They can be a sequence of expressions separated with the *";"* lexeme.
- They can be annotated with a type by using the *":"* lexeme.

Before we begin a deeper discussion on bindings, note that F# adheres to two different syntaxes: *verbose* and *lightweight*. In the verbose syntax, newlines and whitespaces are generally ignored, while in lightweight syntax, certain keywords and lexemes may be replaced by newlines and whitespaces. The lightweight syntax is the most common and will be used in this book.

4.1 Value Bindings

Binding identifiers to literals, or expressions that are evaluated to be values, is called *value-binding*, and examples are `let a = 3.0` and `let b = cos 0.9`. Value bindings have the following syntax:

Listing 4.2: Value binding expression.

```
let <valueIdent> = <bodyExpr>
```

The `let` keyword binds a value-identifier with an expression. The above notation means that `<valueIdent>` is to be replaced with a name and `<bodyExpr>` with an expression that evaluates to a value. The binding *is* available in later lines until the end of the scope it is defined in.

The value identifier is annotated with a type by using the *":"* lexeme followed by the name of a type, e.g., `int`. The "_" lexeme may be used as a value-identifier. This lexeme is called the *wildcard pattern*, and for value-bindings it means that `<bodyExpr>` is evaluated, but the result is discarded.

For example, letting the identifier `p` be bound to the value `2.0` and using it in an expression is done as shown in Listing 4.3. Note that the expression `3.0 ** p` must

Listing 4.3 letValueLightWeight.fsx: Lightweight syntax does not require the `in` keyword, but the expression must be aligned with the `let` keyword.

```
let p = 2.0
do printfn "%A" (3.0 ** p)
```

```
$ dotnet fsi letValueLightWeight.fsx
9.0
```

be put in parentheses here, to force F# to *first* calculate the result of the expression and *then* give it to `printfn` to be printed on the screen. The same expression in interactive mode will also show with the inferred types, as shown in Listing 4.4. By

Listing 4.4: Interactive mode also outputs inferred types.

```
> let p = 2.0
do printfn "%A" (3.0 ** p);;
9.0
val p: float = 2.0
val it: unit = ()
```

the `val` keyword in the line `val p : float = 2.0`, we see that `p` is inferred to be of type `float` and bound to the value `2.0`. The inference is based on the type of the right-hand-side which is `float`. Identifiers may be defined to have a type using the "`:`" lexeme, but the types on the left-hand-side and right-hand-side of the "=" lexeme must be identical. Mixing types gives an error, as shown in Listing 4.5. Here, the left-hand side is defined to be an identifier of type float, while the right-hand side is a literal of type integer.

A key concept of programming is *scope*. When F# seeks the value bound to a name, it looks left and upward in the program text for its `let`-binding in the present or

Listing 4.5 letValueTypeError.fsx:
Binding error due to type mismatch.

```
let p = 2.0
do printfn "%A" (3 ** p)
```

```
$ dotnet fsi letValueTypeError.fsx


letValueTypeError.fsx(2,18): error FS0001: The type 'int'
   does not support the operator 'Pow'
```

higher scopes. This is called *lexical scope*. Some special bindings are mutable, that is, they can change over time in which case F# uses the *dynamic scope*. This will be discussed in Section 13.1.

Scopes are given levels, and scopes may be nested, where the nested scope has a level one lower than its parent. F# distinguishes between the top and lower levels, and at the top level in the lightweight syntax, redefining values is not allowed, as shown in Listing 4.6. However, using parentheses, we create a *code block*, i.e., a

Listing 4.6 letValueScopeLowerError.fsx:
Redefining identifiers is not allowed in lightweight syntax at top level.

```
let p = 3
let p = 4
do printfn "%A" p;
```

```
$ fsharpc --nologo -a letValueScopeLowerError.fsx

letValueScopeLowerError.fsx(2,5): error FS0037: Duplicate
   definition of value 'p'
```

nested scope, as demonstrated in Listing 4.7. In lower scope-levels, redefining is

Listing 4.7 letValueScopeBlockAlternative3.fsx:
A block may be created using parentheses.

```
let p = 3
(
  let p = 4
  do printfn "%A" p
)
```

```
$ dotnet fsi letValueScopeBlockAlternative3.fsx
4
```

★ allowed but **avoid reusing names unless it's in a deeper scope.** Inside the block in Listing 4.7 we used indentation, which is good practice, but not required here.

Defining blocks is used for controlling the extent of a lexical scope of bindings. For example, bindings inside a nested scope are not available outside, as shown in Listing 4.8. Nesting is a natural part of structuring code, e.g., through function

Listing 4.8 letValueScopeNestedScope.fsx: Bindings inside a scope are not available outside.

```
let p = 3
(
  let q = 4
  do printfn "%A" q
)
do printfn "%A %A" p q
```

```
$ fsharpc --nologo -a letValueScopeNestedScope.fsx

letValueScopeNestedScope.fsx(6,22): error FS0039: The value
   or constructor 'q' is not defined.
```

definitions to be discussed in Section 4.2 and flow control structures to be discussed in Section 13.2. Blocking code by nesting is a key concept for making robust code that is easy to use by others, without the user necessarily needing to know the details of the inner workings of a block of code.

4.2 Function Bindings

A function is a mapping between an input and output domain, as illustrated in Figure Figure 4.1. A key advantage of using functions when programming is that

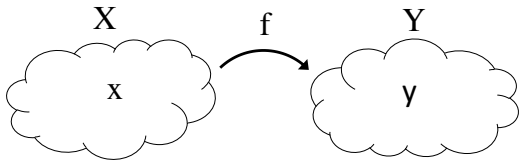


Fig. 4.1 A function $f : X \rightarrow Y$ is a mapping between two domains, such that $f(x) = y$, $x \in X$, $y \in Y$.

they encapsulate code into smaller units, that may be reused and are easier to find errors in. F# is a functional-first programming language and offers a number of alternative methods for specifying parameters, which will be discussed in this section. Binding identifiers to functions follows a syntax similar to value-binding,

Listing 4.9: Function binding expression

```
let <ident> (<arg> {<arg>}) | () =
  <expr>
```

Here `<ident>` is an identifier and is the name of the function, `<arg>` is zero or more identifiers, that bind to the value used when calling the function, and which is to be used in the *body* of the function, the expression `<expr>`. The `|` notation denotes a choice, i.e., either that on the left-hand side or that on the right-hand side. Thus `let f x = x * x` and `let f () = 3` are valid function bindings, but `let f = 3` would be a value binding, not a function binding. The arguments and the function may be annotated with a type, in which case for arguments we write

Listing 4.10: Function binding expression

```
let <ident> ((<arg> : <type>) {(<arg> : <type>)} : <type>) | (
    () : <type>) =
  <expr>
```

where `<type>` is a name of an existing type. The argument types are given in parentheses, and the return type is given last.

Functions are a key concept in F#, and in this chapter, we will discuss the very basics. Recursive functions will be discussed in Section 13.2 and higher-order functions in Chapter 10.

The function's body can be placed either on the same line as the `let`-keyword or on the following lines using indentation. An example of defining a function and using it in interactive mode is shown in Listing 4.11. Here we see that the function is

Listing 4.11: An example of a binding of an identifier and a function.

```

> let sum (x : float) (y : float) : float = x + y
let c = sum 357.6 863.4
do printfn "%A" c;;
1221.0
val sum: x: float -> y: float -> float
val c: float = 1221.0
val it: unit = ()
```

interpreted to have the type `val sum : x:float -> y:float -> float`. The "->" lexeme means a mapping between sets, in this case, floats. The function is also a higher-order function, to be discussed in detail below, and here it suffices to think of `sum` as a function that takes 2 floats as arguments and returns a float.

Not all types need to be declared, just a sufficient number for F# to be able to infer the types for the full statement. For the example, one is sufficient, and we could just have declared the type of the result, as in Listing 4.12. Or even just one of the arguments, as in Listing 4.13. In both cases, since the + *operator* is only defined for

Listing 4.12 letFunctionAlterantive.fsx:
Not every type needs to be declared.

```
let sum x y : float = x + y
```

Listing 4.13 letFunctionAlterantive2.fsx:
Just one type is often enough for F# to infer the rest.

```
let sum (x : float) y = x + y
```

operands of the same type, declaring the type of either arguments or result implies the type of the remainder.

Arguments need not always be inferred to types, but may be of the generic type when *type safety* is ensured, as shown in Listing 4.14. Here, the function `second`

Listing 4.14: Type safety implies that a function will work for any type.

```
> let second x y = y
let a = second 3 5
do printfn "%A" a
let b = second "horse" 5.0
do printfn "%A" b;;
5
5.0
val second: x: 'a -> y: 'b -> 'b
val a: int = 5
val b: float = 5.0
val it: unit = ()
```

does not use the first argument `x`, which therefore can be of any type, and which F#, therefore, calls `'a`. The type of the second element, `y`, can also be of any type and not necessarily the same as `x`, so it is called `'b`. Finally, the result is the same type as `y`, whatever it is. This is an example of a *generic function*, since it will work on any type.

A function may contain a sequence of expressions but must return a value. E.g., the quadratic formula may be written as shown in Listing 4.15. Here, we use a set of nested scopes. In the scope of the function `solution` there are further two functions defined, each with their own scope, which F# identifies by the level of indentation. The amount of space used for indentation does not matter, but all lines in the same scope must use the same amount. The scope ends before the first line with the previous indentation or none. Notice how the last expression is not bound to an identifier but is the result of the function, i.e., in contrast to many other languages, F# does not have an explicit keyword for returning values but requires a final expression, which will be returned to the caller of the function. Note also that since the function `discriminant` is defined in the nested scope of `solution`, and because the scope ends before `let a = 1.0`, the function `discriminant` cannot be called outside `solution`.

Listing 4.15 identifiersExampleAdvance.fsx:
A function may contain sequences of expressions.

```
let solution a b c sgn =
  let discriminant a b c =
    b ** 2.0 - 4.0 * a * c
  let d = discriminant a b c
  (-b + sgn * sqrt d) / (2.0 * a)

let a = 1.0
let b = 0.0
let c = -1.0
let xp = solution a b c +1.0
let xn = solution a b c -1.0
do printfn "0 = %A * x ** 2.0 + %A * x + %A" a b c
do printfn "  has solutions %A and %A" xn xp
```

```
$ dotnet fsi identifiersExampleAdvance.fsx
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has solutions -1.0 and 1.0
```

Lexical scope and function definitions can be a cause of confusion, as the following example in Listing 4.16 shows. Here, the value-binding for `a` is redefined after it

Listing 4.16 lexicalScopeNFunction.fsx:
Lexical scope means that $f(z) = 3x$ and not $4x$ at the time of calling.

```
let testScope x =
  let a = 3.0
  let f z = a * z
  let a = 4.0
  f x
do printfn "%A" (testScope 2.0)
```

```
$ dotnet fsi lexicalScopeNFunction.fsx
6.0
```

has been used to define a helper function `f`. So which value of `a` is used when we later apply `f` to an argument? To resolve the confusion, remember that value-binding is lexically defined, i.e., the binding `let f z = a * z` uses the value of `a` as it is defined by the ordering of the lines in the script, not dynamically by when `f` was called. Hence, **think of lexical scope as substitution of an identifier with its value or function immediately at the place of definition.** Since `a` and `3.0` are synonymous in the first lines of the program, the function `f` is really defined as `let f z = 3.0 * z`. ★

Functions do not need a name, but may be declared as an *anonymous function* using the *fun* keyword and the "->" lexeme, as shown in Listing 4.17. Here, a name is bound to an anonymous function which returns the first of two arguments.

Listing 4.17 functionDeclarationAnonymous.fsx: Anonymous functions are functions as values.

```
let first = fun x y -> x
do printfn "%d" (first 5 3)
```

```
$ dotnet fsi functionDeclarationAnonymous.fsx
5
```

The difference to `let first x y = x` is that anonymous functions may be treated as values, meaning that they may be used as arguments to other functions and the new values may be reassigned to their identifiers when mutable, as will be discussed in Section 13.1. A common use of anonymous functions is as arguments to other functions, as demonstrated in Listing 4.18. Note that here `apply` is given

Listing 4.18 functionDeclarationAnonymousAdvanced.fsx: Anonymous functions are often used as arguments for other functions.

```
let apply f x y  = f x y
let mul = fun a b -> a * b
do printfn "%d" (apply mul 3 6)
```

```
$ dotnet fsi functionDeclarationAnonymousAdvanced.fsx
18
```

3 arguments: the function `mul` and 2 integers. It is not given the result of `mul` ★ `3 6`, since that would not match the definition of `apply`. **Anonymous functions and functions as arguments are powerful concepts but tend to make programs harder to read, and their use should be limited.**

The result of one function is often used as an argument of another. This is function composition, and an example is shown in Listing 4.19. In the example we combine

Listing 4.19 functionComposition.fsx: Composing functions using intermediate bindings.

```
let f x = x + 1
let g x = x * x

let a = f 2
let b = g a
let c = g (f 2)
do printfn "a = %A, b = %A, c = %A" a b c
```

```
$ dotnet fsi functionComposition.fsx
a = 3, b = 9, c = 9
```

two functions `f` and `g` by storing the result of `f 2` in `a` and using that as argument

of `g`. This is the same as `g (f 2)`, and in the later case, the compile creates a temporary value for `f 2`. Such compositions are so common in F# that a special set of operators has been invented, called the *piping* operators: "`|>`" and "`<|`". They are used as demonstrated in Listing 4.20. The example shows regular composition,

Listing 4.20 functionPiping.fsx:
Composing functions by piping.

```
1 let f x = x + 1
2 let g x = x * x
3
4 let a = g (f 2)
5 let b = 2 |> f |> g
6 let c = g <| (f <| 2)
7 do printfn "a = %A, b = %A, c = %A" a b c
```

```
1 $ dotnet fsi functionPiping.fsx
2 a = 9, b = 9, c = 9
```

left-to-right, and right-to-left piping. The word piping is a pictorial description of data as if it were flowing through pipes, where functions are connection points of pipes distributing data in a network. The three expressions in Listing 4.20 perform the same calculation. The left-to-right piping in line 5 corresponds to the left-to-right reading direction, i.e., the value `2` is used as argument to `f`, and the result is used as argument to `g`. In contrast, right-to-left piping in line 6 has the order of arithmetic composition as line 4. Unfortunately, since the piping operators are left-associative, without the parenthesis in line 6 `g <| f <| 2`, F# would read the expression as `(g <| f) <| 2`. That would have been an error since `g` takes an integer as an argument, not a function. F# can also define composition on a function level. Further discussion on this is deferred to Chapter 10.

A *procedure* is a generalization of the concept of functions, and in contrast to functions, procedures need not return values. This is demonstrated in Listing 4.21. In F#, this is automatically given the unit type as the return value. Procedural

Listing 4.21 procedure.fsx:
A procedure is a function that has no return value, and in F# returns "()".

```
1 let printIt a = printfn "This is '%A'" a
2 do printIt 3
3 do printIt 3.0
```

```
1 $ dotnet fsi procedure.fsx
2 This is '3'
3 This is '3.0'
```

thinking is useful for *encapsulation* of scripts, but is prone to *side-effects*. For this

★ reason, it is advised to **prefer functions over procedures.** More on side-effects in Section 13.1.

In F#, functions (and procedures) are *first-class citizens*, which means that functions are values: They may be passed as arguments, returned from a function, and bound to a name. For first-class citizens, the name it is bound to does not carry significance to the language, as, e.g., illustrated with the use of anonymous functions. Technically, a function is stored as a *closure*. A closure is a description of the function, its arguments, its expression, and the environment at the time it was created, i.e., the triple $(args, exp, env)$. Consider the listing in Listing 4.22. It defines two functions

Listing 4.22 functionFirstClass.fsx:
The function `ApplyFactor` has a non-trivial closure.

```
1 let mul x y = x * y
2 let factor = 2.0
3 let applyFactor fct x =
4   let a = fct factor x
5   string a
6
7 do printfn "%g" (mul 5.0 3.0)
8 do printfn "%s" (applyFactor mul 3.0)
```

```
1 $ dotnet fsi functionFirstClass.fsx
2 15
3 6
```

`mul` and `applyFactor`, where the latter is a higher-order function taking another function as an argument and using part of the environment to produce its result. The two closures are:

$$\texttt{mul} : (\text{args}, \text{exp}, \text{env}) = \big((\texttt{x}, \texttt{y}), (\texttt{x * y}), ()\big) \tag{4.3}$$

$$\texttt{applyFactor} : (\text{args}, \text{exp}, \text{env}) = ((\texttt{x}, \texttt{fct}), (\texttt{body}), (\texttt{factor} \rightarrow 2.0)) \tag{4.4}$$

where lazily write `body` instead of the whole function's body. The function `mul` does not use its environment, and everything needed to evaluate its expression is values for its arguments. The function `applyFactor` also takes two arguments, a function and a value. It uses `factor` from the environment, thus this is stored in its closure. When `mul` is given as an argument in Listing 4.22 line 8, then it is its closure which is given to `applyFactor`, and the closure contains everything that `applyFactor` requires to use `mul`. Likewise, if `applyFactor` is given as the argument to yet another function, then its closure includes the relevant part of its environment at the time of definition, `factor`, such that when `applyFactor` is applied to two arguments, then its closure contains everything needed to evaluate its expression.

4.3 Do-Bindings

Aside from `let`-bindings that bind names with values or functions, sometimes we just need to execute code. This is called a *do*-binding or, alternatively, a *statement*. The syntax is as follows:

Listing 4.23: Syntax for `do`-bindings.

```
[do ]<expr>
```

The expression `<expr>` must return `unit`. The keyword `do` is optional in most cases, but using it emphasizes that the expression is not a function that returns a useful value. Procedures are examples of such expressions, and a very useful family of procedures is the `printf` family described below. In the remainder of this book, we will refrain from using the `do` keyword.

4.4 Conditional Expressions

Programs often contain code that should only be executed under certain conditions. The `match` – `with` expressions allows us to write such expressions, and its syntax is as follows:

Listing 4.24: Syntax for `match`-expressions.

```
match <inputExpr> with
 [| ]<pat> [when <guardExpr>] -> <caseExpr>
 | <pat> [when <guardExpr>] -> <caseExpr>
 | <pat> [when <guardExpr>] -> <caseExpr>
 ...
```

where `<inputExpr>` is the *input pattern* to find matches of, `<pat>` is a pattern to match with, `<guardExpr>` is an optional guard expression, and `<caseExpr>` is the resulting expression. Each set starting with `<pat>` is called a case. The indentation must be equal to or higher than the indentation of `match`. All cases must return a value of the same type, and F# reports an error when the complete domain of the input pattern is not covered by cases in `match`-expressions.

For example, a calendar program may need to highlight weekends, and thus highlighting code should be called, if and only if the day of the week is either Saturday og Sunday, as shown in the following. Remember that `match` – `with` is the last expression in the function `whatToDoToday`, hence the resulting string of `match` – `with` is also the resulting string of the function. F# examines each case from top to bottom, and as soon as a matching case is found, then the code following the

Listing 4.25 weekdayPatternSimple.fsx: Using `match` – `with` to print discriminated unions.

```
let whatToDoToday (d : int) : string =
  match d with
    0 -> "go to work"
    | 1 -> "go to work"
    | 2  -> "go to work"
    | 3 -> "go to work"
    | 4 -> "go to work"
    | 5 -> "take time off"
    | 6  -> "take time off"
    | _ -> "unknown day of the week"

let dayOfWeek = 3
let task = whatToDoToday dayOfWeek
printfn "Day %A of the week you should %A" dayOfWeek task
```

```
$ dotnet fsi weekdayPatternSimple.fsx
Day 3 of the week you should "go to work"
```

"->"-symbol is executed. This code is called the case's *branch*. The "_" pattern is a *wildcard* and matches everything.

In the above code, each day has its own return string associated with it, which is great, if we want to return different messages for different days. However, in this code, we only return two different strings, and if, e.g., choose the one for weekdays, then we need to make 5 identical changes to the code. This is wasteful and risks introducing new errors. So instead, we may compact this code by concatenating the "|"'s as shown in Listing 4.26. It still seems unnecessarily clumsy to have to

Listing 4.26 weekdayPatternSum.fsx: Concatenating cases. Compare with Listing 4.25.

```
let whatToDoToday (d : int) : string =
  match d with
    0 | 1 | 2  | 3 | 4 -> "go to work"
    | 5 | 6  -> "take time off"
    | _ -> "unknown day of the week"

let dayOfWeek = 3
let task = whatToDoToday dayOfWeek
printfn "Day %A of the week you should %A" dayOfWeek task
```

```
$ dotnet fsi weekdayPatternSum.fsx
Day 3 of the week you should "go to work"
```

mention the number of each weekday explicitly and to avoid this, we may use *guards* as illustrated in Listing 4.27. In the guard expression, we match d with the name

Listing 4.27 weekdayPatternGuard.fsx:
Using guards. Compare with Listing 4.26.

```
let whatToDoToday (d : int) : string =
  match d with
    n when 0 <= n && n < 5 -> "go to work"
    | n when 5 <= n && n <= 6  -> "take time off"
    | _ -> "unknown day of the week"

let dayOfWeek = 3
let task = whatToDoToday dayOfWeek
printfn "Day %A of the week you should %A" dayOfWeek task
```

```
$ dotnet fsi weekdayPatternGuard.fsx
Day 3 of the week you should "go to work"
```

`n` as long as the condition is fulfilled, i.e., as long $0 \leq n < 5$ or as it is written in F#, `0 <= n && n < 5`. The logical and-operator ("&&") is needed in F#, since both `0 <= n` and `n < 5` needs to be true, and F# can only evaluate them individually.

Note that `match` – `with` expressions will give a warning if there are no cases for the full domain of what is being matched. I.e., if we don't end with the wildcard pattern when matching for integers, we would be missing cases for all other patterns

Listing 4.28 weekdayPatternError.fsx:
Using `match` – `with` to print discriminated unions.

```
let whatToDoToday (d : int) : string =
  match d with
    n when 0 <= n && n < 5 -> "go to work"
    | n when 5 <= n && n <= 6  -> "take time off"

let dayOfWeek = 3
let task = whatToDoToday dayOfWeek
printfn "Day %A of the week you should %A" dayOfWeek task
```

```
$ dotnet fsi weekdayPatternError.fsx


weekdayPatternError.fsx(2,9): warning FS0025: Incomplete
   pattern matches on this expression.

Day 3 of the week you should "go to work"
```

An alternative to `match` – `with` is the `if` – `then` expressions.

Listing 4.29: Conditional expressions.

```
if <cond> then <expr> {elif <cond> then <expr>} [else <expr>]
```

The condition <con> is an expression resulting in a Boolean value, and there can be zero or more `elif` conditions, as indicated by {}. Each expression <expr> is called a *branch*, and all branches must have the same type, such that regardless of which branch is chosen, the type of the result of the conditional expression is the same. Then the expression of the first if-branch, whose condition is true, is evaluated. If all conditions are false then the `else`-branch is evaluated. If no `else` expression is present, then "()" will be returned. See Listing 4.30 for a simple example. The branches are often several lines of code, in which case it is more

Listing 4.30 weekdayIfThenElse.fsx:
Conditional computation with `if` – `then`. Compare with Listing 4.27.

```
let whatToDoToday (d : int) : string =
    if 0 <= d && d < 5 then "go to work"
    elif 5 <= d && d <= 6  then "take time off"
    else "unknown day of the week"

let dayOfWeek = 3
let task = whatToDoToday dayOfWeek
printfn "Day %A of the week you should %A" dayOfWeek task
```

```
$ dotnet fsi weekdayIfThenElse.fsx
Day 3 of the week you should "go to work"
```

useful to write them indented, below the keywords, which may also make the code easier to read. The identical `whatToDoToday` function with indented branches is shown in Listing 4.31. In contrast to `match` – `with`, the `if` – `then` expression is

Listing 4.31 weekdayIfThenElseIndentation.fsx:
Conditional computation with `if` – `then`. Compare with Listing 4.27.

```
let whatToDoToday (d : int) : string =
    if 0 <= d && d < 5 then
        "go to work"
    elif 5 <= d && d <= 6  then
        "take time off"
    else
        "unknown day of the week"
```

not as thorough in investigating whether cases of the domain in question have been covered. For example, in both

```
let a = if true then 3
```

and

```
let a = if true then 3 elif false then 4
```

the value of `a` is uniquely determined, but F# finds them to be erroneous since F# looks for the `else` to ensure all cases have been covered, and that the branches have the identical type. Hence,

```
let a = if true then 3 else 4
```

is the only valid expression of the 3. In practice, F# assumes that the omitted branch returns "`()`", and thus it is fine to say `let a = if true then ()` and `if true then printfn "hej"`. Nevertheless, it is good practice in F# to always include an `else` branch.

4.5 Tracing code by hand

The concept of Tracing by hand will be developed throughout this book. Here we will concentrate on the basics, and as we introduce more complicated programming structures, we will develop the Tracing by hand accordingly. Tracing may seem tedious in the beginning but in conjunction with strategically placed debugging `printfn` statements, it is a very valuable tool for debugging.

Consider the program in Listing 4.32. The program calls `testScope 2.0`, and by

Listing 4.32 lexicalScopeTracing.fsx:
Example of lexical scope and closure environment.

```
let testScope x =
  let a = 3.0
  let f z = a * z
  let a = 4.0
  f x
printfn "%A" (testScope 2.0)
```

```
$ dotnet fsi lexicalScopeTracing.fsx
6.0
```

running the program, we see that the return-value is `6.0` and not `8.0`, as we had expected. Hence, we will use tracing to understand the result.

Tracing a program by hand means that we simulate its execution and, as part of that, keep track of the bindings, assignments closures, scopes, and input and output of the program. To do this, we need to consider the concept of *environments*.

Environments describe bindings available to the program at the present scope and at a particular time and place in the code. There is always an outer environment, called E_0, and each time we call a function or create a scope, we create a new environment. Only one environment can be active at a time, and it is updated as we simulate the execution of code with new bindings and temporary evaluations of expressions. Once a scope is closed, then its environment is deleted and a return value is transported to its enclosing environment. In tracing, we note return values explicitly. Likewise, output from, e.g., `printfn` is reported with a special notation.

To trace code, we make a table with 4 columns: Step, Line, Environment, and Bindings and evaluations. The Step column enumerates the steps performed. The Line column contains the program-line treated in a step *where* the present environment is updated. The Environment contains the name of the present environment, and Bindings . . . shows *what* in the environment is updated.

The code in Listing 4.32 contains a function definition and a call, hence, the first lines of our table look like this,

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	testScope = $\big((x)$, testScope-body, $()\big)$
2	6	E_0	testScope 2.0 = ?

The elements of the table are to be understood as follows. Step 0 initializes the outer environment. In order for us to remember that the environment is empty, we write the unit value "`()`". Reading the code from top to bottom, the first nonempty and non-comment line we meet is line 1, hence, in Step 1, we update the environment with the binding of a function to the name `testScope`. Since functions are values in F#, we note their bindings by their closures: a list of argument names, the function-body, and the values lexically available at the place of binding. See Section 4.2 for more information on closures. Following the function-binding, the `printfn` statement is called in line 6 to print the result `testScope 2.0`. However, before we can produce any output, we must first evaluate `testScope 2.0`. Since we do not yet know what this function evaluates to, in Step 2 we simply write the call with a question mark. The call causes the creation of a new environment, and we continue our table as follows,

Step	Line	Env.	Bindings and evaluations
3	1	E_1	$\big((x = 2.0)$, testScope-body, $()\big)$

This means that we are going to execute the code in testScope-body. The function was called with 2.0 as an argument, causing $x = 2.0$. Hence, the only binding available at the start of this environment is to the name `x`. In the testScope-body, we make 3 further bindings and a function call. First to `a`, then to `f`, then to another `a`, which will overshadow the previous binding, and finally we call `f`. Thus, our table is updated as follows,

Step	Line	Env.	Bindings and evaluations
4	2	E_1	$a = 3.0$
5	3	E_1	f = $\big((z)$, a * z, $(a = 3.0, x = 2.0)\big)$
6	4	E_1	$a = 4.0$
7	5	E_1	f x = ?

Note that by lexical scope, the closure of `f` includes everything above its binding in E_1, and therefore we add $a = 3.0$ and $x = 2.0$ to the environment element in its closure. This has consequences for the following call to `f` in line 5, which creates a new environment based on `f`'s closure and the value of its arguments. The value of `x` in Step 7 is found by looking in the previous steps for the last binding to the name `x` in E_1, which occurs in Step 3. Note that the binding to a name `x` in Step 5 is an internal binding in the closure of `f` and is irrelevant here. Hence, we continue the table as,

Step	Line	Env.	Bindings and evaluations
8	3	E_2	$\big((z = 2.0)$, a * z, $(a = 3.0, x = 2.0)\big)$

Executing the body of `f`, we initially have 3 bindings available: `z = 2.0`, `a = 3.0`, and `x = 2.0`. Thus, to evaluate the expression `a * z`, we use these bindings and write,

Step	Line	Env.	Bindings and evaluations
9	3	E_2	$a * z = 6.0$
10	3	E_2	return = 6.0

The 'return'-word is used to remind us that this is the value to replace the question mark within Step 7. Here we will make a mental note and not physically replace the question mark with the calculated value. If you are ever in doubt which call is connected with which return value, seek upwards in the table from the return statement for the first question mark. Now we delete E_2 and return to the enclosing environment, E_1. Here the function call was the last expression, hence the return-value from `testScope` will be equal to the return-value from `f`, and we write,

Step	Line	Env.	Bindings and evaluations
11	3	E_1	return = 6.0

Similarly, we delete E_1 and return to the question mark in Step 2, which is replaced by the value 6.0. We can now finish the `printfn` statement and produce the output,

Step	Line	Env.	Bindings and evaluations
12	6	E_0	output = "6.0\n"

The return-value of a `printfn` statement is `()`, and since this line is the last of our program, we return `()` and end the program:

Step	Line	Env.	Bindings and evaluations
13	6	E_0	return = ()

The full table is shown for completeness in Table 4.2. Hence, we conclude that the

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	testScope = $\big((x)$, testScope-body, $()\big)$
2	6	E_0	testScope 2.0 = ?
3	1	E_1	$\big((x = 2.0)$, testScope-body, $()\big)$
4	2	E_1	$a = 3.0$
5	3	E_1	f = $\big((z)$, a * z, $(a = 3.0, x = 2.0)\big)$
6	4	E_1	$a = 4.0$
7	5	E_1	f x = ?
8	3	E_2	$\big((z = 2.0)$, a * z, $(a = 3.0, x = 2.0)\big)$
9	3	E_2	$a * z = 6.0$
10	3	E_2	return = 6.0
11	3	E_1	return = 6.0
12	6	E_0	output = "6.0\n"
13	6	E_0	return = ()

Table 4.2 The complete table produced while tracing the program in Listing 4.32 by hand.

program outputs the value `6.0`, since the function `f` uses the first binding of `a = 3.0`, and this is because the binding of `f` to the expression `a * z` creates a closure with a lexical scope. Thus, in spite that there is an overshadowing value of `a`, when `f` is called, this binding is ignored in the body of `f`. To correct this, we update the code as shown in Listing 4.33.

Listing 4.33 lexicalScopeTracingCorrected.fsx:
Tracing the code in Listing 4.32 by hand produced the table in Table 4.2, and to get the desired output, we correct the code as shown here.

```
let testScope x =
  let a = 4.0
  let f z = a * z
  f x
printfn "%A" (testScope 2.0)
```

```
$ dotnet fsi lexicalScopeTracingCorrected.fsx
8.0
```

4.6 Key Concepts and Terms in This Chapter

In this chapter, we have taken the first look at organizing code. Key concepts have been:

- **Binding** values and functions using the **let** statements.
- Function **closures**, which are the values bound in **let** statements of functions.
- Certain names are forbidden in particular **keywords**, which are reserved for other uses.
- In contrast to **let** expressions, **do** statements do something, such as `printfn` prints to the screen. They return the "`()`" value.
- Conditional expressions using **match–with** and some **patterns** and **guards**. Alternatively, **if–then** expressions can be used to a similar effect.
- **Trace by hand** as a method for simulating execution and in particular keeping track of the **environments** defined by the code structure.

Chapter 5

Programming with Types

Abstract In the previous chapter, we took the first step in organizing code such that it better reflects the solutions we seek, is reusable, and is easier to find possible errors in. A fundamental structure in all of this is types, which much like sets in mathematics, form the basic building blocks of what we express in code. In this chapter, we will focus on making new types to better express our solutions. We will examine how to combine types to express higher-order information such as

- defining new types that simultaneously combine existing types.
- define alternatives, i.e., a type that can be one of several other types.

After you have read this chapter, you will be able to model values that contain

- a combination of types such as an address consisting of both street names as a string and zip codes as an integer.
- an alternative list of types such as a shape being either a circle parametrized by its center and radius or square parametrized by its four corners.

F# is a strongly typed language, meaning that types are known or inferred at compile time. In the previous chapters, we have used *primitive types* such as `float` and `bool`, function types, and compound types implicitly defined by tuples. These types are used for simple programming tasks, and everything that can be programmed can be accomplished using these types. However, larger programs are often easier to read and write when using more complicated type structures. In this chapter, we will discuss type abbreviations, enumerated types, discriminated unions, records, and structs. Class types are discussed in depth in Chapter 15.

5.1 Type Products: Tuples

Tuples are a direct extension of constants. They are immutable and have neither concatenations nor indexing operations. Tuples are unions of immutable types and have the following syntax:

Listing 5.1: Tuples are a list of expressions separated by commas.

```
<expr>{, <expr>}
```

Tuples are identified by the "`,`" lexeme and often enclosed in parentheses, but that is not required. An example is a triple, also known as a 3-tuple, `(2,true,"hello")`. In interactive mode, the type of tuples is demonstrated in Listing 5.2. The values

Listing 5.2: Tuple types are products of sets.

```

> let tp = (2, true, "hello")
printfn "%A" tp;;
(2, true, "hello")
val tp: int * bool * string = (2, true, "hello")
val it: unit = ()
```

`2`, `true`, and `"hello"` are *members*, and the number of elements of a tuple is its *length*. From the response of F#, we see that the tuple is inferred to have the type `int * bool * string`. The "`*`" denotes the Cartesian product between sets. Tuples can be products of any type and follow the lexical scope rules like value and function bindings. Notice also that a tuple may be printed as a single entity by the `%A` placeholder. In the example we bound `tp` to the tuple. The opposite is also possible, as demonstrated in Listing 5.3. In this example, a function is defined that takes 1 argument, a 3-tuple. If we wanted a function with 3 arguments, then the function binding should have been `let deconstructNPrint a b c = ...`. The value binding `let (a, b, c) = tp`, binds a tuple with 3 named members to a value, thus deconstructing it in terms of its members. This is called pattern matching. Since we used the `\%A` placeholder in the `printfn` function, the function can be

Listing 5.3: Definition of a tuple.

```
> let deconstructNPrint tp =
    let (a, b, c) = tp
    printfn "tp = (%A, %A, %A)" a b c
deconstructNPrint (2, true, "hello")
deconstructNPrint (3.14, "Pi", 'p');;
tp = (2, true, "hello")
tp = (3.14, "Pi", 'p')
val deconstructNPrint: 'a * 'b * 'c -> unit
val it: unit = ()
```

called with 3-tuples of different types. F# informs us that the tuple type is variable by writing `'a * 'b * 'c`. The "`'`" notation means that the type can be decided at run-time, see Section 5.5 for more on variable types.

Pairs or 2-tuples are so common that F# includes two built-in functions, *`fst`* and *`snd`*, to extract the first and second element of a pair. This is demonstrated in Listing 5.4.

Listing 5.4 pair.fsx:
Deconstruction of pairs with the built-in functions `fst` and `snd`.

```
let pair = ("first", "second")
printfn "fst(pair) = %s, snd(pair) = %s" (fst pair) (snd pair)
```

```
$ fsharpc --nologo pair.fsx && mono pair.exe
fst(pair) = first, snd(pair) = second
```

Tuples of equal lengths can be compared, and the comparison is defined similarly to string comparison. Tuples of equal length are compared element by element. E.g., `(1,2) = (1,3)` is false, while `(1,2) = (1,2)` is true. The "<>" operator is the boolean negation of the "=" operator. For the "<" , "<=", ">", and ">=" operators, the strings are ordered lexicographically, such that `('a', 'b', 'c') < ('a', 'b', 's') && ('a', 'b', 's') < ('c', 'o', 's')` is true, that is, the "<" operator on two tuples is true if and only if the left operand should come before the right when sorting alphabetically. See Listing 5.5 for an example. The algorithm for deciding the boolean value of `(a1, a2) < (b1, b2)` is as follows: we start by examining the first elements, and if `a1` and `b1` are different, then the result of `(a1, a2) < (b1, b2)` is equal to the result of `a1 < b1`. If `a1` and `b1` are equal, then we move on to the next letter and repeat the investigation. The "<=", ">", and ">=" operators are defined similarly.

Binding tuples to mutables does not make the tuple mutable. This is demonstrated in Listing 5.6. However, it is possible to define a mutable variable of type tuple such that new tuple values can be assigned to it, as shown in Listing 5.7. Mutable tuples are value types, meaning that binding to new names makes copies, not aliases, as demonstrated in Listing 5.8. The use of tuples shortens code and highlights semantic

Listing 5.5 tupleCompare.fsx:
Tuples comparison is similar to string comparison.

```
let lessThan (a, b, c) (d, e, f) =
  if a <> d then a < d
  elif b <> e then b < d
  elif c <> f then c < f
  else false

let printTest x y =
  printfn "%A < %A is %b" x y (lessThan x y)

let a = ('a', 'b', 'c');
let b = ('d', 'e', 'f');
let c = ('a', 'b', 'b');
let d = ('a', 'b', 'd');
printTest a b
printTest a c
printTest a d
```

```
$ fsharpc --nologo tupleCompare.fsx && mono tupleCompare.exe
('a', 'b', 'c') < ('d', 'e', 'f') is true
('a', 'b', 'c') < ('a', 'b', 'b') is false
('a', 'b', 'c') < ('a', 'b', 'd') is true
```

Listing 5.6 tupleOfMutables.fsx:
A mutable changes value, but the tuple defined by it does not refer to the new value.

```
let mutable a = 1
let mutable b = 2
let c = (a, b)
printfn "%A, %A, %A" a b c
a <- 3
printfn "%A, %A, %A" a b c
```

```
$ fsharpc --nologo tupleOfMutables.fsx && mono
   tupleOfMutables.exe
1, 2, (1, 2)
3, 2, (1, 2)
```

content at a higher level, e.g., instead of focusing on the elements, tuples focus on their union. While this may look elegant and short there is the risk of *obfuscation*, i.e., writing compact code that is difficult to read, where an unprepared reader of the code may not easily understand the computation nor appreciate its elegance without
★ an accompanying explanation. Hence, **always keep an eye out for compact and concise ways to write code, but never at the expense of readability.**

Listing 5.7 mutableTuple.fsx:
A mutable tuple can be assigned a new value.

```
let mutable pair = 1,2
printfn "%A" pair
pair <- (3,4)
printfn "%A" pair
```

```
$ dotnet fsi mutableTuple.fsx
(1, 2)
(3, 4)
```

Listing 5.8 mutableTupleValue.fsx:
A mutable tuple is a value type.

```
let mutable pair = 1,2
let mutable aCopy = pair
pair <- (3,4)
printfn "%A %A" pair aCopy
```

```
$ dotnet fsi mutableTupleValue.fsx
(3, 4) (1, 2)
```

5.2 Type Sums: Discriminated Unions

A discriminated union is a union of a set of named cases. These cases can further be of specified types. The syntax for defining a discriminated union is as follows:

Listing 5.9: Syntax for type abbreviation.

```
[<attributes>]
type <ident> =
  [| ]<ident> [of [<ident> :] <aType> [* [<ident> :] <aType>
   ...]]
  | <ident> [of [<ident> :] <aType> [* [<ident> :] <aType>
   ...]]
  ...
```

Discriminated unions are reference types, i.e., their content is stored on *The Heap*. Since they are immutable, there is no risk of side-effects. As reference types, they only pass a reference when used as arguments to and returned from a function. This is in contrast to value types, which transport a complete copy of the data structure. Discriminated unions are thus effective for large data structures. Discriminated unions can also be represented as structures using the `[<Struct>]` attribute, in which case they are value types.

An example just using the named cases but no further specification of types is given in Listing 5.10. Here, we define a discriminated union as three named cases

Listing 5.10 discriminatedUnions.fsx:
A discriminated union of medals.

```
1 type medal =
2   Gold
3   | Silver
4   | Bronze
5
6 let aMedal = medal.Gold
7 printfn "%A" aMedal
```

```
1 $ fsharpc --nologo discriminatedUnions.fsx && mono
     discriminatedUnions.exe
2 Gold
```

signifying three different types of medals. A commonly used discriminated union is the *option type*.

Discriminated unions may also be used to store data. Where the names in enumerated types are aliases of single values, the names used in discriminated unions can hold any value specified at the time of creation. An example is given in Listing 5.11. In this case, we define a discriminated union of two and three-dimensional vectors.

Listing 5.11 discriminatedUnionsOf.fsx:
A discriminated union using explicit subtypes.

```
1 type vector =
2   Vec2D of float * float
3   | Vec3D of x : float * y : float * z : float
4
5 let v2 = Vec2D (1.0, -1.2)
6 let v3 = Vec3D (x = 1.0, z = -1.2, y = 0.9)
7 printfn "%A and %A" v2 v3
```

```
1 $ fsharpc --nologo discriminatedUnionsOf.fsx && mono
     discriminatedUnionsOf.exe
2 Vec2D (1.0, -1.2) and Vec3D (1.0, 0.9, -1.2)
```

Values of these types are created using their names followed by a tuple of their arguments. The names are also called field names. The field names may be used when creating discriminated union values, as shown in Line 6. When used, then the arguments may be given in arbitrary order, nevertheless, values for all fields must be given.

5.3 Records

A record is a compound of named values, and a record type is defined as follows:

Listing 5.12: Syntax for defining record types.

```
[ <attributes> ]
type <ident> = {
  [ mutable ] <label1> : <type1>
  [ mutable ] <label2> : <type2>
  ...
}
```

Records are collections of named variables and values of possibly different types. They are reference types, and thus their content is stored on *The Heap*. Records can also be *struct records* using the `[<Struct>]` attribute, in which case they are value types. An example of using records is given in Listing 5.13. The values of individual members of a record are obtained using the *"."* notation This example illustrates

Listing 5.13 records.fsx:
A record is defined as holding information about a person.

```
type person = {
    name : string
    age : int
    height : float
}

let author = {name = "Jon"; age = 50; height = 1.75}
printfn "%A\nname = %s" author author.name
```

```
$ fsharpc --nologo records.fsx && mono records.exe
{ name = "Jon"
  age = 50
  height = 1.75 }
name = Jon
```

how record type is used to store varied data about a person.

If two record types are defined with the same label set, then the latter dominates the former. This is demonstrated in Listing 5.14. In the example, two identical record types are defined, and we use the built-in `GetType()` method to inspect the type of bindings. We see that `lecturer` is of `RecordsDominance+teacher` type, since `teacher` dominates the identical `person` type definition. However, we may enforce the `person` type by either specifying it for the name, as in `let author : person = ...`, or by fully or partially specifying it in the record expression following the "=" sign. In both cases, they are of `RecordsDominance+person` type. The built-in

Listing 5.14 recordsDominance.fsx:
Redefined types dominate old record types, but earlier definitions are still accessible using the explicit or implicit specification for bindings.

```
1  type person = { name : string; age : int; height : float }
2  type teacher = { name : string; age : int; height : float }
3
4  let lecturer = {name = "Jon"; age = 50; height = 1.75}
5  printfn "%A : %A" lecturer (lecturer.GetType())
6  let author : person = {name = "Jon"; age = 50; height = 1.75}
7  printfn "%A : %A" author (author.GetType())
8  let father = {person.name = "Jon"; age = 50; height = 1.75}
9  printfn "%A : %A" author (author.GetType())
```

```
1  $ fsharpc --nologo recordsDominance.fsx && mono
      recordsDominance.exe
2  { name = "Jon"
3    age = 50
4    height = 1.75 } : RecordsDominance+teacher
5  { name = "Jon"
6    age = 50
7    height = 1.75 } : RecordsDominance+person
8  { name = "Jon"
9    age = 50
10   height = 1.75 } : RecordsDominance+person
```

`GetType()` method is inherited from the base class for all types, see Chapter 15 for a discussion on classes and inheritance.

Note that when creating a record you must supply a value to all fields, and you cannot refer to other fields of the same record, i.e., `{name = "Jon"; age = height * 3; height = 1.75}` is illegal.

Since records are per default reference types, binding creates aliases, not copies. This matters for mutable members, in which case when copying, we must explicitly create a new record with the old data. Copying can be done either by using referencing the individual members of the source or using the short-hand *`with`* notation. This is demonstrated in Listing 5.15. Here, `age` is defined as a mutable value and can be changed using the usual "<-" assignment operator. The example demonstrates two different ways to create records. Note that when the mutable value `author.age` is changed in line 10, then `authorAlias` also changes, since it is an alias of `author`, but neither `authorCopy` nor `authorCopyAlt` changes, since they are copies. As illustrated, copying using `with` allows for easy copying and partial updates of another record value.

Listing 5.15 recordCopy.fsx:
Bindings are references. To copy and not make an alias, explicit copying must be performed.

```
type person = {
  name : string;
  mutable age : int;
}

let author = {name = "Jon"; age = 50}
let authorAlias = author
let authorCopy = {name = author.name; age = author.age}
let authorCopyAlt = {author with name = "Noj"}
author.age <- 51
printfn "author : %A" author
printfn "authorAlias : %A" authorAlias
printfn "authorCopy : %A" authorCopy
printfn "authorCopyAlt : %A" authorCopyAlt
```

```
$ fsharpc --nologo recordCopy.fsx && mono recordCopy.exe
author : { name = "Jon"
  age = 51 }
authorAlias : { name = "Jon"
  age = 51 }
authorCopy : { name = "Jon"
  age = 50 }
authorCopyAlt : { name = "Noj"
  age = 50 }
```

5.4 Type Abbreviations

F# allows for renaming of types, which is called *type abbreviation* or *type aliasing*. The syntax is:

Listing 5.16: Syntax for type abbreviation.

```
type <ident> = <aType>
```

where the identifier is a new name, and the type-name is an existing type or a compound of existing types. Listing 5.17 shows examples of the definition of several type abbreviations. Here we define the abbreviations `size`, `position`, `person`, and `intToFloat`, and later make bindings enforcing the usage of these abbreviations.

Type abbreviations are used as short abbreviations of longer types, and they add semantic content to the program text, thus making programs shorter and easier to read. Type abbreviations allow the programmer to focus on the intended structure at a higher level by, e.g., programming in terms of a type `position` rather than `float * float`. Thus, they often result in programs with fewer errors. Type abbreviations

Listing 5.17 typeAbbreviation.fsx:
Defining four type abbreviations, three of which are compound types.

```
type size = int
type position = float * float
type person = string * int
type intToFloat = int -> float

let sz : size = 3
let pos : position = (2.5, -3.2)
let pers : person = ("Jon", 50)
let conv : intToFloat = fun a -> float a
printfn "%A, %A, %A, %A" sz pos pers (conv 2)
```

```
$ fsharpc --nologo typeAbbreviation.fsx && mono
   typeAbbreviation.exe
3, (2.5, -3.2), ("Jon", 50), 2.0
```

also make maintenance easier. For instance, if we at a later stage decide that positions can only have integer values, then we only need to change the definition of the type abbreviation, not every place a value of type `position` is used.

5.5 Variable Types

An advanced topic in F# is *variable types*. There are three different versions of variable types in F#: *runtime resolved*, which have the syntax `'<ident>`, *anonymous*, which are written as *"_"*, and *statically resolved*, which have the syntax `^<ident>`. Variable types are particularly useful for functions that work for many types. An example of a generic function and its use is given in Listing 5.18. In this example, the

Listing 5.18 variableType.fsx:
A function `apply` with runtime resolved types.

```
let apply (f : 'a -> 'a -> 'a) (x : 'a) (y : 'a) : 'a = f x y
let intPlus (x : int) (y : int) : int = x + y
let floatPlus (x : float) (y : float) : float = x + y

printfn "%A %A" (apply intPlus 1 2) (apply floatPlus 1.0 2.0)
```

```
$ fsharpc --nologo variableType.fsx && mono variableType.exe
3 3.0
```

function `apply` has runtime resolved variable type, and it accepts three parameters: `f`, `x`, and `y`. The function will work as long as the parameters for `f` is a function of

two parameters of identical type, and `x` and `y` are values of the same type. Thus, in the `printfn` statement we are able to use `apply` for both an integer and a float variant.

The example in Listing 5.18 illustrates a very complicated way to add two numbers. The "+" operator works for both types out of the box, so why not something simpler like relying on the F# type inference system by not explicitly specifying types, as attempted in Listing 5.19? Unfortunately, the example fails to compile, since the type

Listing 5.19 variableTypeError.fsx:
Even though the "+" operator is defined for both integers and floats, the type inference is static and infers `plus : int -> int`.

```
let plus x y = x + y

printfn "%A %A" (plus 1 2) (plus 1.0 2.0)
```

```
$ fsharpc --nologo variableTypeError.fsx && mono
   variableTypeError.exe

variableTypeError.fsx(3,34): error FS0001: This expression
   was expected to have type
    'int'
but here has type
    'float'

variableTypeError.fsx(3,38): error FS0001: This expression
   was expected to have type
    'int'
but here has type
    'float'
```

inference is performed at compile time, and by `plus 1 2`, it is inferred that `plus : int -> int`. Hence, calling `plus 1.0 2.0` is a type error. Function bindings allow for the use of the *`inline`* keyword, and adding this successfully reuses the definition of `plus` for both types, as shown in Listing 5.20. In the example, adding

Listing 5.20 variableTypeInline.fsx:
The keyword `inline` forces static and independent inference each place the function is used. Compare to the error case in Listing 5.19.

```
let inline plus x y = x + y

printfn "%A %A" (plus 1 2) (plus 1.0 2.0)
```

```
$ fsharpc --nologo variableTypeInline.fsx && mono
   variableTypeInline.exe
3 3.0
```

the `inline` does two things: Firstly, it copies the code to be performed to each place the function is used, and secondly, it forces statically resolved variable type checking independently in each place. The type annotations inferred as a result of the `inline`-keyword may be written explicitly, as shown in Listing 5.21. The example

Listing 5.21 compiletimeVariableType.fsx:
Explicitly spelling out of the statically resolved type variables from Listing 5.19.

```
let inline plus (x : ^a) (y : ^a) : ^a when ^a : (static
   member ( + ) :  ^a * ^a -> ^a) = x + y

printfn "%A %A" (plus 1 2) (plus 1.0 2.0)
```

```
$ fsharpc --nologo compiletimeVariableType.fsx && mono
   compiletimeVariableType.exe
3 3.0
```

in Listing 5.21 demonstrates the statically resolved variable type syntax, `^<ident>`, as well as the use of *type constraints*, using the keyword *when*. Type constraints have a rich syntax, but will not be discussed further in this book. In the example, the type constraint `when ^a : (static member ( + ) :  ^a * ^a -> ^a)` is given using the object-oriented properties of the type variable `^a`, meaning that the only acceptable type values are those which have a member function `(+)` taking a tuple and giving a value all of the identical types, and where the type can be inferred at compile time. See Chapter 15 for details on member functions.

The `inline` construction is useful when generating generic functions and still profiting from static type checking. However, explicit copying of functions is often something better left to the compiler to optimize. An alternative seems to be using runtime resolved variable types with the `'<ident>` syntax. Unfortunately, this is not possible in the case of most operators, since they have been defined in the `FSharp.Core` namespace to be statically resolved variable types. E.g., the "+" operator has type `( + ) : ^T1 -> ^T2 -> ^T3 (requires ^T1 with static member (+) and ^T2 with static member (+))`.

Discrimate Unions and type abbreviations can be generic as well. For example, in Listing 5.22, we demonstrate how an option-like wrapper can be made for any type. As shown here, the variable type `'a` is first fixed, when a value of the `myOption` is created, and in contrast to function types that are statically resolved, the same definition can be reused for different types of discriminated unions. Similarly, in Listing 5.23, we give an example of a variable type abbreviation. Here, the variable type is a function, which takes a list of some type and returns an value of the same type. For example, `head` returns the first element of any list type, and `avg` returns the average value of lists of floats. For a more in-depth example of generic types, see Section 9.5.

Listing 5.22: A variable discrimated union.

```
> type myOption<'a> = Value of 'a | Error
let intOption = Value 1
let charOption = Value 'a';;
type myOption<'a> =
  | Value of 'a
  | Error
val intOption: myOption<int> = Value 1
val charOption: myOption<char> = Value 'a'
```

Listing 5.23: A variable type abreviation.

```
> type summarize<'a> = 'a list -> 'a
let head : summarize<'a> = List.head
let avg : summarize<float> = List.average
let lst = [0.0..3.0]
printfn "head lst = %A" (head lst)
printfn "avg lst = %A" (avg lst);;
head lst = 0.0
avg lst = 1.5
type summarize<'a> = 'a list -> 'a
val head: summarize<'a>
val avg: summarize<float>
val lst: float list = [0.0; 1.0; 2.0; 3.0]
val it: unit = ()
```

5.6 Key Concepts and Terms in This Chapter

In this chapter you have learned about:

- the **product type** also known as a **tuple**, which is equivalent to a set product;
- the **sum type** also known as a **discriminate union**;
- the **records** which are similar to tuples, but allows you to name the entries
- and as an advanced topic, you have seen F# has flexibility in specifying types either at compile or runtime.

Chapter 6

Making Programs and Documenting Them

Abstract Programs are more than a set of instructions, which when executed produces the desired result. Programming is an activity, and a program is the result of a process, in which a problem has been expressed, analyzed, subdivided, implemented, tested, and possibly rephrased. And often the process and its result are to be wrapped and documented for it to be useful by the programmer or others. In this chapter, we will zoom out, and focus on some of these surrounding processes. The chapter will describe:

- How to design functions.
- How and why to document programs using in-code documentation.

6.1 The 8-step Guide to Writing Functions

Pólya's problem-solving technique described in Section 1.2 is a useful starting point for solving problems, and for the object-oriented programming paradigm to be discussed in later chapters Chapter 17, approaches such as Pólya's have been put into systems. Regardless of the origin, there is always a point, where a programmer has to focus on small-scale problems such as: Which functions, should be used, and how should the functions be designed? This is not an area heavily investigated in the literature, but often a skill that programmers pick up by actively engaging in programming alone or with other programmers. However, here I will venture a recipe for designing functions, which I and my colleagues call The 7-step guide to writing functions. It is not meant as the ultimate guide or as required steps, but it is our experience that these steps contain essential elements that consciously or perhaps unconsciously always take part in designing useful and reusable functions.

To decide which functions to write and how to write them:

1. Note: Write a short note on what a function should do.
2. Name: Invent a name for the function. Semantically meaningful names should be preferred.
3. External test: Write a small test program, which uses the yet-to-be-written function.
4. Type: Decide what type, the function should have, e.g., by how you used it in your test program.
5. Implement: Write the function and possibly its helper functions.
6. Internal test: Extend your test program with more examples, where you use the function and based on its implementation.
7. Run: Run the test program
8. Document: Write brief in-code documentation of the function, see Section 6.2.

As an example, let us revisit the problem of solving a quadratic equation: The task is to find the zero-crossings of a second-degree polynomial, i.e., $f(x) = ax^2+bx+c = 0$. The process could be as follows:

1. Note: We decide to stick to the mathematical description:

   > Given parameters a, b, and c, the function should return the 0, 1, or possibly 2 locations x, where $f(x) = 0$.

2. Name: This function may be used together with solvers for other equations, so we decide to give it the rather long name

   `solveQuadraticEquation`.

3. External test: We decide on a single test to get a feeling of how the function is to be used:

Listing 6.1: Defining the function sum

```
let p = solveQuadraticEquation 1.0 0.3 -1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %1" p;;
```

4. Type: Since there may be 0-2 points x, where $f(x) = 0$, the output answers could be a tuple. Further, since $a, b, c, x \in \mathbb{R}$, we will use floats. Hence,

   `solveQuadraticEquation -> float -> float -> float -> float*float`.

5. Implement: Thinking about how to write `solveQuadraticEquation`, we decide that since the calculation of the discriminant is done twice, we will add it as a helper function. Our resulting code is:

Listing 6.2: Defining the function sum

```
let discriminant a b c = b ** 2.0 - 4.0 * a * c

let solveQuadraticEquation a b c =
  let d = discriminant a b c
  ((-b + sqrt d) / (2.0 * a),
   (-b - sqrt d) / (2.0 * a))
```

6. Internal test: Working with the code, we realize that it is unclear, what happens, when there are 0 og 1 solutions, so we update the external test and add more tests:

Listing 6.3: Defining the function sum

```
let p1 = solveQuadraticEquation 1.0 0.3 -1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p1
let p2 = solveQuadraticEquation 1.0 0.0 0.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p2
let p3 = solveQuadraticEquation 1.0 0.0 1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p3
```

7. Run: The complete code with examples and its output, when executed is shown in Listing 6.4

Listing 6.4 solveQuadraticEquation.fsx: Solving quadratic equations

```
let discriminant a b c = b ** 2.0 - 4.0 * a * c

let solveQuadraticEquation a b c =
  let d = discriminant a b c
  ((-b + sqrt d) / (2.0 * a),
   (-b - sqrt d) / (2.0 * a))

let p1 = solveQuadraticEquation 1.0 0.3 -1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p1
let p2 = solveQuadraticEquation 1.0 0.0 0.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p2
let p3 = solveQuadraticEquation 1.0 0.0 1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p3
```

```
$ dotnet fsi solveQuadraticEquation.fsx
0=1.0x^2+0.3x-1.0 => x = (0.8611874208, -1.161187421)
0=1.0x^2+0.3x-1.0 => x = (0.0, -0.0)
0=1.0x^2+0.3x-1.0 => x = (nan, nan)
```

8. Document: The following section will discuss how to perform in-code documentation and use Listing 6.4 as an example.

6.2 Programming as a Communication Activity

Documentation is a very important part of writing programs since it is most unlikely that you will be writing really obvious code. Moreover, what seems obvious at the point of writing may be mystifying months later to the author and to others. Documentation serves several purposes:

1. Communicate to the user of the code, what it does and how to use it. In this book, we will emphasize the XML-standard for this purpose.

2. Highlight big insights essential for the code, which is important for other programmers to understand and maintain the code.

3. Highlight possible conflicts and/or areas where the code could be changed later, which is also targeted programmers rather than users of the code.

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. Documentation is most often a mixture of in-code documentation and accompanying

documents. Here, we will focus on in-code documentation which arguably causes problems in multi-language environments and run the risk of bloating code. Since documentation is about human-to-human communication, there is no correct documentation. However, as in all things, documentation can both be too little and too much, and the ability to produce documentation is best learned by example and by doing.

F# has two different syntaxes for comments. Comments can be block comments:

Listing 6.5: Block comments.

```
1 (*<any text>*)
```

The comment text `(<any text>)` can be any text and is stilled parsed by F# as keywords and basic types, implying that `(* a comment (* in a comment *) *)` and `(* "*)" *)` are valid comments, while `(* " *)` is invalid.

Alternatively, comments may also be line comments,

Listing 6.6: Line comments.

```
1 //<any text>
```

where the comment text ends after the first newline.

The block and line comments are used principally for communicating insights and comments into the code between programmers who want to understand and/or maintain the code.

Users of the code, are most likely also programmers but have an outside perspective. They are more interested in what the code does, and how it is to be used. For this we recommend the *Extensible Markup Language* documentation standard (*XML-standard*) [1]. All lines of the XML-standard start with a triple-slash `///`. Thus, it is a line-comment, where an extra slash has been added for visual flair. XML consists of tags which always appear in pairs, e.g., the tag "tag" would look like `<tag> ... </tag>`. A subset of tags are listed in Table 6.1. If no tags are used, then it is automatically assumed to be a `<summary>`. An example of a documented script is shown in Listing 6.7. is:

Several tools exist that extract the comments from source code and reorder the comments into manual type structures, such as Doxygen. Popular output from such tools is both HTML and LaTeX. However, for this text, the usage of the XML-standard as a way to standardize comments will suffice.

[1] For specification of C# documentations comments see ECMA-334: `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`

Tag	Description
`<c>`	Set text in a code-font.
`<code>`	Set one or more lines in code-font.
`<example>`	Set as an example.
`<exception>`	Describe the exceptions a function can throw.
`<list>`	Create a list or table.
`<para>`	Set text as a paragraph.
`<param>`	Describe a parameter for a function or constructor.
`<paramref>`	Identify that a word is a parameter name.
`<permission>`	Document the accessibility of a member.
`<remarks>`	Further describe a function.
`<returns>`	Describe the return value of a function.
`<see>`	Set as link to other functions.
`<seealso>`	Generate a See Also entry.
`<summary>`	Main description of a function or value.
`<typeparam>`	Describe a type parameter for a generic type or method.
`<typeparamref>`	Identify that a word is a type parameter name.
`<value>`	Describe a value.

Table 6.1 Recommended XML tags for documentation comments, from ECMA-334 3rd Edition, Annex E, Section 2.

6.3 Key Concepts and Terms in This Chapter

This chapter has considered elements that are an important part of the activity of programming, but to some extent complement the specific act of writing source code. You have seen:

- How to use the **7-step guide** to design functions, which emphasizes writing examples of function usage before implementing the function itself.
- Write **in-code** documentation to support the understanding of the code.
- Documentation is written for programmers and there are at least two different types: **users** and **maintainers**.
- The **XML standard** uses `///`, is for both types of programmers, and documents what a program does and how it is to be used.
- The **line** and **block** comments are for implementation-specific details and intended to be read by programmers who seek to understand and maintain the code.
- There is no such thing as the correct documentation, but you are well advised to follow the XML standard and to improve your skill by writing documentation and sharing it with others.

Listing 6.7 commentExample.fsx:
Code with XML comments.

```
/// The discriminant of a quadratic equation with
/// parameters a, b, and c
let discriminant a b c = b ** 2.0 - 4.0 * a * c

/// <summary>Find x when 0 = ax^2+bx+c.</summary>
/// <remarks>Negative discriminants are not checked.</remarks>
/// <example>
///   The following code:
///   <code>
//      let p = solveQuadraticEquation 1.0 0.3 -1.0
///     printfn "0=1.0x^2+0.3x-1.0 => x = %A" p
///   </code>
///   prints <c>0=1.0x^2+0.3x-1.0 => x = (0.9, -1.2)</c>.
/// </example>
/// <param name="a">Quadratic coefficient.</param>
/// <param name="b">Linear coefficient.</param>
/// <param name="c">Constant coefficient.</param>
/// <returns>The solution to x as a tuple.</returns>
let solveQuadraticEquation a b c =
  let d = discriminant a b c
  ((-b + sqrt d) / (2.0 * a),
   (-b - sqrt d) / (2.0 * a))

let p1 = solveQuadraticEquation 1.0 0.3 -1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p1
let p2 = solveQuadraticEquation 1.0 0.0 0.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p2
let p3 = solveQuadraticEquation 1.0 0.0 1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p3
```

Part II

Declarative Programming Paradigms

A programming problem may have many solutions, e.g., squaring a real value, x^2, can in F# be written as both `x*x` and `x**2.0`, and more complicated problems typically have many valid solutions. Particularly long programs can be complex and can have a high risk of programming errors. Different programming languages offer different structures to aid the programming in managing complex solutions, which is sometimes called a *programming paradigm*. Paradigms may be classified as either *declarative* or *imperative*. Some languages such as F# are multiparadigm, making the boundary between programming paradigms fuzzy, however, in their pure form, programs in declarative programming languages are a list of properties of the desired result without a specification on how to compute it, while programs in imperative programming languages is a specific set of instructions on how to change *states* on the computer in order to reach the desired result. This part will emphasize the *functional programming paradigm*, which is a declarative paradigm. In Part III, the *imperative* and the *object-oriented programming paradigms* will be emphasized.

Functional programming is a style of programming which performs computations by evaluating functions. Functional programming is declarative in nature, e.g., by the use of value- and function-bindings – `let`-bindings – and avoids statements – `do`-bindings. Thus, all values are constants, and the result of a function in functional programming depends only on its arguments. It is deterministic, i.e., repeated call to a function with the same arguments always gives the same result. In functional programming, data and functions are clearly separated, and hence data structures are dum as compared to objects in object-oriented programming paradigm, see Part III. Functional programs clearly separate behavior from data and subscribes to the view that *it is better to have 100 functions operate on one data structure than 10 functions on 10 data structures*. Simplifying the data structure has the advantage that it is much easier to communicate data than functions and procedures between programs and environments. The .Net, mono, and java's virtual machine are all examples of an attempt to rectify this, however, the argument still holds.

The functional programming paradigm can trace its roots to lambda calculus introduced by Alonzo Church in 1936 [1]. Church designed lambda calculus to discuss computability. Some of the forces of the functional programming paradigm are that it is often easier to prove the correctness of code, and since no states are involved, then functional programs are often also much easier to parallelize than other paradigms.

Functional programming has a number of features:

Pure functions
: Functional programming is performed with pure functions. A pure function always returns the same value, when given the same arguments, and it has no side-effects. A function in F# is an example of a pure function. Pure functions can be replaced by their result without changing the meaning of the program. This is known as *referential transparency*.

higher-order functions
: Functional programming makes use of higher-order functions, where functions may be given as arguments and returned as results of a function application. higher-order functions and *first-class citizenship* are related concepts, where higher-order functions are the mathematical description of functions that operator on functions, while a first-class citizen is the computer science term for functions as values. F# implements higher-order functions. The `List.map` is and example of a higher-order functions.

Recursion
: Functional programs use recursion instead of `for`- and `while`-loops. Recursion can make programs ineffective, but compilers are often designed to optimize *tail-recursion* calls. Common recursive programming structures are often available as optimized higher-order functions such as *iter*, *map*, *reduce*, *fold*, and *foldback*. F# has good support for all of these features.

Immutable states
: Functional programs operate on values, not on *mutable values* also known as *variables*. This implies *lexicographical scope* in contrast to mutable values, which implies *dynamic scope*.

Strongly typed
: Functional programs are often strongly typed, meaning that types are set no later than at *compile-time*. F# does have the ability to perform runtime type assertion, but for most parts it relies on explicit *type annotations* and *type inference* at compile-time. This means that type errors are caught at compile time instead of at runtime.

Lazy evaluation
: Due to referential transparency, values can be computed any time up until the point when it is needed. Hence, they need not be computed at compilation time, which allows for infinite data structures. F# has support for lazy evaluations using the `lazy`-keyword, sequences using the `seq`-type, and computation expressions, all of which are advanced topics and not treated in this book.

Immutable states imply that data structures in functional programming are different than in imperative programming. E.g., in F# lists are immutable, so if an element of a list is to be changed, a new list must be created by copying all old values except that which is to be changed. Such an operation is therefore linear in computational complexity. In contrast, arrays are mutable values, and changing a value is done by reference to the value's position and changing the value at that location. This has constant computational complexity. While fast, mutable values give dynamic scope and makes reasoning about the correctness of a program harder, since mutable states do not have referential transparency.

Functional programming may be considered a subset of *imperative programming*, in the sense that functional programming does not include the concept of a state, or one may think of functional programming as only having one unchanging state. Functional programming also has a bigger focus on declaring rules for *what* should be solved, and not explicitly listing statements describing *how* these rules should be combined and executed in order to solve a given problem. Functional programming is often found to be less error-prone at runtime, making more stable, safer programs that are less open for, e.g., hacking.

Chapter 7

Lists

Abstract In programming, a list is an abstract data type, which contains a list of elements, such as a list of shopping items Figure 7.1, a list of students in a class, and a to-do list. Cornerstones in functional programming are immutable values and

Fig. 7.1 A list of shopping items.

functions, and in the previous chapters, we have looked at many ways where this is sufficient for solving many problems. However, often data is on the form of a list in which equivalent expressions need to be calculated and possibly collected into a single value. For example, for the shopping list, we may want to estimate the total shopping price by 1) replacing each item on the list with a price, and 2) summing the elements. In functional programming, this can be done with the programming concept of map and fold, where map produces a new list as the elements of the original list with a function applied to them, and fold sequentially combines the values of the

price-list by iteratively adding the price to a subtotal. These are important examples of the usage of lists, but there is more. In this chapter, you will learn how to

- define lists
- manipulate lists using indexing and its properties
- use the list module including the map and fold higher-order functions.

After you have read this chapter, you will be able to model situations such as

- a class with lists of student records.
- a drawing consisting of a list of elements, such as a house, some trees, etc.

Lists are unions of immutable values of the same type. A list can be expressed as a *sequence expression*,

Listing 7.1: The syntax for a list using the sequence expression.

```
1 [[<expr>{; <expr>}]]
```

For example, `[1; 2; 3]` is a list of integers, `["This"; "is"; "a"; "list"]` is a list of strings, `[(fun x -> x); (fun x -> x*x)]` is a list of functions, and `[]` is the empty list. Lists may also be given as ranges,

Listing 7.2: The syntax for a list using the range expressions.

```
1 [<expr> .. <expr> [.. <expr>]]
```

where `<expr>` in *range expressions* must be of integers, floats, or characters. Examples are `[1 .. 5]`, `[-3.0 .. 2.0]`, and `['a' .. 'z']`. Range expressions may include a step size, thus, `[1 .. 2 .. 10]` evaluates to `[1; 3; 5; 7; 9]`.

F# implements lists as linked lists, as illustrated in Figure 7.2. A linked list is a data

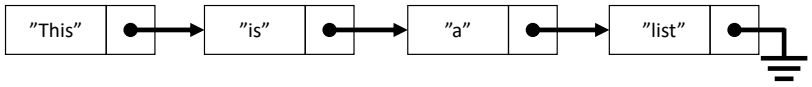


Fig. 7.2 A list is a linked list: Here is illustrated the linked list of `["This"; "is"; "a"; "list"]`.

structure consisting of a number of elements, where each element has an item of data and a pointer to the next element. For lists, every element can only have one other element pointing to it. The first element in a list is called its *head*, and the remainder of the list is called its *tail*. The last element in a list does not point to any other, and this is denoted by the *ground* symbol as shown in the figure.

A list type is identified with the `list` keyword, such that a list of integers has the type `int list`. Like strings, lists may be indexed and sliced using the *“[]”* notation, and the first element has index 0, and the last has the list's length minus one, see Listing 7.3 for examples. Note that if the index must be positive and less that than the length of the list. Otherwise an *out-of-bounds exception* is cast. See Section 12.3 for more details on exceptions. This is a very typical error, and it is adviced to **program in ways which completely avoids the possibility of out-of-bounds indexing, e.g., by using the List module.** An alternative to the “`[]`” indexing notation is the `List.tryItem` function to be described below on page 105, which does not cast exceptions but returns and option type. ⋆

Listing 7.3 listIndexing.fsx:
Lists are indexed as strings and has a `Length` property.

```
let lst = [3..9]
printfn "lst = %A, lst[1] = %A" lst lst[1]
printfn "First 2 elments of lst = %A" lst[..1]
printfn "Last 3 elements of lst = %A" lst[4..]
printfn "Element number 3 to 5 = %A" lst[2..4]
printfn "All elements = %A" lst[*]
```

```
$ dotnet fsi listIndexing.fsx
lst = [3; 4; 5; 6; 7; 8; 9], lst[1] = 4
First 2 elments of lst = [3; 4]
Last 3 elements of lst = [7; 8; 9]
Element number 3 to 5 = [5; 6; 7]
All elements = [3; 4; 5; 6; 7; 8; 9]
```

A list has a number of *properties*, which is summarized below:

`Head`: Returns the first element of a non-empty list.

Listing 7.4: `Head`

```
> [1; 2; 3].Head;;
val it: int = 1
```

Hence, given a non-empty list `lst`, then `lst.Head = lst[0]`.

`IsEmpty`: Returns true if the list is empty and false otherwise.

Listing 7.5: `IsEmpty`

```
> [1; 2; 3].IsEmpty;;
val it: bool = false
```

Hence, given a list `lst`, then `lst.IsEmpty` is the same as `lst = []`.

`Length`: Returns the number of elements in the list.

Listing 7.6: `Length`

```
> [1; 2; 3].Length;;
val it: int = 3
```

`Tail`: Returns the list, except for its first element. The list must be non-empty.

Listing 7.7: `Tail`

```
> [1; 2; 3].Tail;;
val it: int list = [2; 3]
```

Hence, given a non-empty list `lst`, then `lst.Tail=lst[1..]`.

A new list may be generated by concatenated two other lists using *concatenation* operator, "`@`". Alternatively, a new list may be generated by prepending an element using the *cons* operators, "`::`". This is demonstrated in Listing 7.8. Since lists

Listing 7.8: Examples of list concatenation.

```
> [1] @ [2; 3];;
val it: int list = [1; 2; 3]

> [1; 2] @ [3; 4];;
val it: int list = [1; 2; 3; 4]

> 1 :: [2; 3];;
val it: int list = [1; 2; 3]
```

are represented as linked lists, some operations on lists are slow and some are fast. Operations on data structures are often analyzed for their *computational complexity*, which is a worst-case and relative measure of their running time on a given piece of hardware. The notation is sometimes called *Big-O* notation or *asymptotic notation*. For example, the algorithm for calculating the length of a linked list starts at the head and follows the links until the end. For a list of length n, this takes n steps, and hence the computational complexity $O(n)$. Conversely, the cons operator is very efficient and has computational complexity $O(1)$, since we only need to link a single element to the head of a list. Another example is concatenation which has computational complexity $O(n)$, where n is the length of the first list. A final example, indexing an element i of a list of length n is also $O(n)$, since in the worst case, $i = n$, and the linked list must be traversed from the head to its tail.

Technically speaking, if the true running time as a function of the length of the list is $f(n)$, then its computational complexity is $O(g(n))$, if there exists a positive real number A and a real number $n_0 > 0$ such that for all $n \leq n_0$,

$$f(n) \leq Ag(n). \tag{7.1}$$

I.e., if the computational complexity is $O(n)$, then for sufficiently large n, the running time grows no faster than Mn for some constant M. This constant will differ per computer, but the asymptotic notation describes the relative increase in running time as we increase the list size.

Pattern matching in the `match–with` expression is possible with combination of the "`[]`" and "`::`" notations in a manner similar to indexing and prepending elements.

For example, recursively iterating over list is often done as illustrated in Listing 7.9. The `match–with` expression recognizes explict naming of elements such as 3-

Listing 7.9 listRecursive.fsx:
Using `match–with` to recursively print the elements of a list.

```
let rec printList (lst : int list) : unit =
  match lst with
    [] ->
      printfn ""
    | elm::rest ->
      printf "%A " elm
      printList rest

printList [3; 4; 5]
```

```
$ dotnet fsi listRecursive.fsx
3 4 5
```

element list `[s;t;u]`, and the cons-notation `head::tail`, where `head` is the first element of a non-empty list, and `tail` is the possibly empty tail. In particular, patterns for single element list can either be `head::[]`, `[head]`, or `s when s.Length = 1`.

It is possible to make multidimensional lists as lists of lists, as shown in Listing 7.10. The example shows a *ragged multidimensional list*, since each row has a different

Listing 7.10 listMultidimensional.fsx:
A ragged multidimensional list, built as lists of lists, and its indexing.

```
let a = [[1;2];[3;4;5]]
let row =  a[0]
let elm = a[0][1]
printfn "Fst row = %A, snd element in fst row = %A" row elm
```

```
$ dotnet fsi listMultidimensional.fsx
Fst row = [1; 2], snd element in fst row = 2
```

number of elements. This is also illustrated in Figure 7.3.

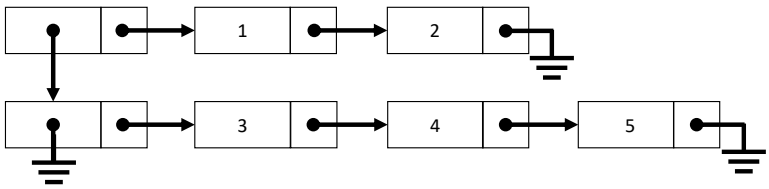


Fig. 7.3 A list is a ragged linked list: Here is illustrated the linked list of `[[1;2];[3;4;5]]`.

7.1 The List Module

Lists are so commonly used that a *module* is included with F#. A module is a library of functions and will be discussed in general in Section 9.2. The list of functions in the list module is very long, and here, we briefly showcase some important ones. For the full list, see `https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-listmodule.html`. Below, the functions are grouped into 3: Simple functions, which take and/or give lists as argument and/or results; higher-order functions, which takes a function as an argument; and those that mimic the list's properties which can be useful in conjunction with the higher-order functions. Higher-order functions are discussed in detail in Chapter 10.

Note that some arguments may result in an error, for example, when trying to find a non-existing element. For this, the list module has two types of functions: Those that return an option type, e.g., `None` if the element sought is not in the list. These functions all start with `try`. Similar functions exists with names excluding `try`, and they cast an exception, in case of an error. See Section 12.3 for more on exceptions. In both cases, the error has to be handled, and in this book, we favor option types.

Some often used simple functions in alphabetical order are:

`List.tryLast`: `'T list -> 'T option`.
Returns the last element of a list as an option type.

Listing 7.11: `List.tryLast`

```
> List.tryLast [1; -2; 0];;
val it: int option = Some 0
```

`List.rev`: `lst:'T list -> 'T list`.
Returns a new list with the same elements as in `lst` but in reversed order.

Listing 7.12: `List.rev`

```
> List.rev [1; 2; 3];;
val it: int list = [3; 2; 1]
```

`List.sort`: `lst:'T list -> 'T list`.
Returns a new list with the same elements as in `lst` but where the elements are sorted.

Listing 7.13: `List.sort`

```
> List.sort [3; 1; 2];;
val it: int list = [1; 2; 3]
```

`List.unzip`: `lst:('T1 * 'T2) list -> 'T1 list * 'T2 list`.
Returns a pair of lists of all the first elements and all the second elements of `lst`, respectively.

Listing 7.14: `List.unzip`

```
> List.unzip [(1, 'a'); (2, 'b'); (3, 'c')];;
val it: int list * char list = ([1; 2; 3], ['a'; 'b';
   'c'])
```

There exists an equivalent function `List.unzip3`, which separates elements from lists of triples.

`List.zip`: `lst1:'T1 list -> lst2:'T2 list -> ('T1 * 'T2) list`.
Returns a list of pairs, where elements in `lst1` and `lst2` are iteratively paired.

Listing 7.15: `List.zip`

```
> List.zip [1; 2; 3] ['a'; 'b'; 'c'];;
val it: (int * char) list = [(1, 'a'); (2, 'b'); (3, 'c')]
```

There exists an equivalent function `List.zip3`, which combines elements form three lists.

Some programming patterns on lists involve performing calculations and combining list elements, and they are so common that they have been standardized. Examples of this are the higher-order functions from the module, given below. As is common, the examples are given with anonymous functions, see page 55 in Section 4.2.

`List.exists`: `f:('T -> bool) -> lst:'T list -> bool`.
Returns true if `f` is true for some element in `lst` and otherwise false.

Listing 7.16: `List.exists`

```
> List.exists (fun x -> x % 2 = 1) [0 .. 2 .. 4];;
val it: bool = false
```

`List.filter`: `f:('T -> bool) -> lst:'T list -> 'T list`.
Returns a new list with all the elements of `lst` for which `f` evaluates to true.

Listing 7.17: `List.filter`

```
> List.filter (fun x -> x % 2 = 1) [0 .. 9];;
val it: int list = [1; 3; 5; 7; 9]
```

`List.fold`: `f:('S -> 'T -> 'S) -> acc:'S -> lst:'T list -> 'S`.
Updates an accumulator iteratively by applying `f` to each element in `lst`. The initial value of the accumulator is `elm`. For example, when `lst` consists of `n` elements `List.fold` calculates:

`f (...(f (f elm lst[0]) lst[1])...) lst[n-1]`.

Listing 7.18: `List.fold`

```
> let addSquares acc elm = acc + elm*elm
List.fold addSquares 0 [0 .. 9];;
val addSquares: acc: int -> elm: int -> int
val it: int = 285
```

There exists an equivalent function `List.fold2`, which iterates through two lists simultaneously.

`List.foldBack`: `f:('T -> 'S -> 'S) -> lst:'T list -> acc:'S -> 'S`.
Updates an accumulator iteratively backwards by applying `f` to each element in `lst`. The initial value of the accumulator is `elm`. For example, when `lst` consists of `n` elements `List.foldBack` calculates:

`f lst[0] (f lst[1] (...(f lst[n-1] elm)...))`.

Listing 7.19: `List.foldBack`

```
> let addSquares elm acc = acc + elm*elm
List.foldBack addSquares [0 .. 9] 0;;
val addSquares: elm: int -> acc: int -> int
val it: int = 285
```

There exists an equivalent function `List.foldBack2`, which iterates through two lists simultaneously.

`List.forall`: `f:('T -> bool) -> lst:'T list -> bool`.
Returns true if all elements in `lst` are true when `f` is applied to them.

Listing 7.20: `List.forall`

```
> List.forall (fun x -> x % 2 = 1) [0 .. 9];;
val it: bool = false
```

There exists an equivalent function `List.forall2`, which iterates through two lists simultaneously.

`List.init`: `m:int -> f:(int -> 'T) -> 'T list`.
Create a list with `m` elements, and whose value is the result of applying `f` to the index of the element.

Listing 7.21: `List.init`

```
> List.init 10 (fun i -> i * i);;
val it: int list = [0; 1; 4; 9; 16; 25; 36; 49; 64; 81]
```

`List.iter`: `f:('T -> unit) -> lst:'T list -> unit`.
Applies `f` to every element in `lst`.

Listing 7.22: `List.iter`

```
> List.iter (fun x -> printfn "%A " x) [0; 1; 2];;
0
1
2
val it: unit = ()
```

There exists an equivalent function `List.iter2`, which iterates through two lists simultaneously, and `List.iteri` and `List.iteri2`, which also receives the index while iterating.

`List.map`: `f:('T -> 'U) -> lst:'T list -> 'U list`.
Returns a list as a concatenation of applying `f` to every element of `lst`.

Listing 7.23: `List.map`

```
> List.map (fun x -> x*x) [0 .. 9];;
val it: int list = [0; 1; 4; 9; 16; 25; 36; 49; 64; 81]
```

There exist equivalent functions `List.map2` and `List.map3`, which iterates through two and three lists simultaneously, and `List.mapi` and `List.mapi2`, which also receives the index while iterating.

`List.tryFind`: `f:('T -> bool) -> lst:'T list -> 'T option`.
Returns the first element of `lst` for which `f` is true as an option type.

Listing 7.24: `List.tryFind`

```
> List.tryFind (fun x -> x % 2 = 1) [0 .. 2 .. 9];;
val it: int option = None
```

`List.tryFindIndex`: `f:('T -> bool) -> lst:'T list -> int option`.
Returns the index of the first element of `lst` for which `f` is true as an option type.

Listing 7.25: `List.tryFindIndex`

```
> List.findIndex (fun x -> x = 'k') ['a' .. 'z'];;
val it: int = 10
```

At times, e.g., in conjunction with the higher-order functions given above, it is useful to have the list operators and properties on function form. These are available in the list module as:

`List.concat`: `lstLst:'T list list -> 'T list`.
Concatenates a list of lists.

Listing 7.26: `List.concat`

```
> List.concat [[1; 2]; [3; 4]; [5; 6]];;
val it: int list = [1; 2; 3; 4; 5; 6]
```

`List.isEmpty`: `lst:'T list -> bool`.
Returns true if `lst` is empty.

Listing 7.27: `List.isEmpty`

```
> List.isEmpty [1; 2; 3];;
val it: bool = false
```

`List.length`: `lst:'T list -> int`.
Returns the length of the list.

Listing 7.28: `List.length`

```
> List.length [1; 2; 3];;
val it: int = 3
```

`List.tryHead`: `lst:'T list -> 'T option`.
Returns the first element in `lst` as an option type. .

Listing 7.29: `List.tryHead`

```
> List.tryHead [1; -2; 0];;
val it: int option = Some 1
```

`List.tryItem`: `i:int -> lst:'T list -> 'T option`.
Returns the *i*'th element of a list as an option type.

Listing 7.30: `List.tryItem`

```
> List.tryItem 10 [0..3];;
val it: int option = None
```

E.g., given a non-empty list `lst`, `Some lst.Head = List.tryHead lst` and `Some lst[2] = List.tryItem 2 lst`. Note, the module does not contain an equivalent of `lst.Tail`.

7.2 Programming Intermezzo: Word Statistics

Natural language processing is the field of studying the natural language. This is a vast field and has seen considerable breakthroughs in our understanding of how humans communicate through writing. A common view of texts is as lists of words, so let us consider one of the most basic questions, the study of natural language may ask:

Problem 7.1

What is the longest word in Hans Christian Andersen's fairy tale "The emperor's new clothes"?

The fairy tale, as published online on `https://www.gutenberg.org` in English contains 1885 words. For the sake of demonstration, we will just consider the first seven words,

> "Many years ago, there was an Emperor,. . . "

and ignore punctuation. In such a small example, we quickly realize, that the answer should be "Emperor" which consists of 7 characters, but if we were to analyze the whole text, then the answer would not be as readily found by eye. Thus, we will write a program. To begin, we create a list of the relevant words. In Chapter 12, we will discuss, how to read from a file, but here, we will enter them by hand:

```
let wLst = ["Many";"years";"ago";"there";"was";"an";"Emperor"]
```

Strings have the `Length` property, which if we can read it of each string in the list, then we can decide, which is the longest. The `List.map` function allows us to make a new list as a copy of the old, but where each element `elm` is result of `f elm` for some function `f`. Thus, we make an anonymous function `fun w -> (w,w.Length)` and apply it to every element by `List.map`:

```
let wLen = List.map (fun (w: string) -> (w,w.Length)) wLst
```

Note that we chose to make a list of pairs, such that the word and its length are joined into a single data structure to safeguard us from possibly future mix-ups of words and lengths. What remains is to find the longest. For this, we must traverse, and while doing so, we must maintain an accumulator containing the longest word, we have seen so far. This is a `List.fold` operation. `List.fold` traverses from the head and applies a function to update the accumulator given the present state of the accumulator and the next element. An function for this is `maxWLen`,

```
let maxWLen acc elm = if snd acc > snd elm then acc else elm
```

The initial value of the accumulator must be sensible, in the case that the list is empty, and which we are sure will be disregarded as soon as it is compared to any word. Here, such a value is `("",0)`. Finally, we are ready to call `List.fold`:

```
let longest = List.fold maxWLen ("",0) wLen
```

Thus, the problem is solved with the program shown in Listing 7.31.

Listing 7.31 longestWord.fsx:
Using the list module to find the longest word in an H.C. Andersen fairy tale.

```
let wLst = ["Many";"years";"ago";"there";"was";"an";"Emperor"]
let wLen = List.map (fun (w: string) -> (w,w.Length)) wLst
let maxWLen acc elm = if snd acc > snd elm then acc else elm
let longest = List.fold maxWLen ("",0) wLen
printfn "The longest word is %A" longest
```

```
$ dotnet fsi longestWord.fsx
The longest word is ("Emperor", 7)
```

7.3 Key concepts and terms in this chapter

In this chapter, you have read about

- How to create lists with the "`[]`" notation
- That lists are implemented as linked elements which implies that prepending is fast, but concatenation and indexing are slow.
- The list properties such as `Head`, `Tail`, and `Length`
- The list module with important higher-order functions such as `List.map` and `List.fold`.

Chapter 8

Recursion Revisited

Abstract Recursion is a central concept in functional programming and is used to control flow. A recursive function must obey the 3 laws of recursion:

1. The function must call itself.
2. It must have a base case.
3. The base case must be reached at some point, in order to avoid an infinite loop

In Figure 8.1 is an illustration of the concept of an infinite loop with recursion. In

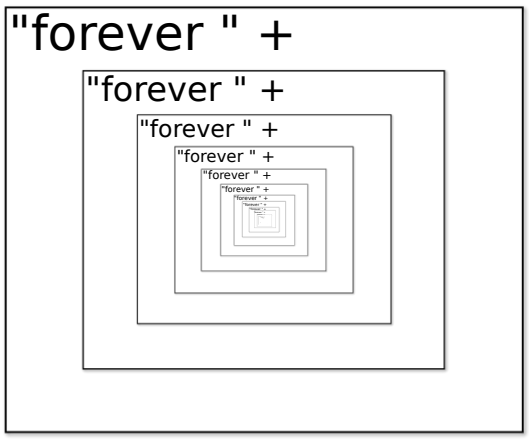


Fig. 8.1 An infinitely long string of "forever forever forever...", conceptually calculated by `let rec forever () = "forever " + (forever ()).`

this chapter, you will learn

- How to define recursive functions and types
- About the concept of the Call stack and tail recursion
- How to trace-by-hand recursive functions

8.1 Recursive Functions

A *recursive function* is a function that calls itself, and the syntax for defining recursive functions is an extension of that for regular functions:

Listing 8.1: Syntax for defining one or more mutually dependent recursive functions.

```
let rec <ident> (<arg> {<arg>}) | () =
  <expr>
{and let <ident> (<arg> {<arg>}) | () =
  <expr>}
```

From a compiler point of view, the *rec* is necessary, since the function is used before the compiler has completed its analysis. If two functions are mutually recursive, i.e., they call each other, then they must be defined jointly using the *and* keyword.

An example of a recursive function that counts from 1 to 10 is given in Listing 8.2. Here the `count` function calls itself repeatedly, such that the first call is `count 1 10`, which calls `count 2 10`, and so on until the last call `count 11 10`. Each time `count` is called, new bindings named `a` and `b` are made to new values. This is illustrated in Figure 8.2. The old values are no longer accessible, as indicated by subscripts in the figure. E.g., in $\texttt{count}_3$, the scope has access to $\texttt{a}_3$ but not $\texttt{a}_2$ and $\texttt{a}_1$. Thus, in this program, the process is similar to a `while` loop, where the counter is `a`, and in each loop its value is reduced.

Listing 8.2 countRecursive.fsx: Counting to 10 using recursion.

```
let rec count a b =
  match a with
    i when i > b ->
      printfn ""
    | _ ->
      printf "%d " a
      count (a + 1) b

count 1 10
```

```
$ dotnet fsi countRecursive.fsx
1 2 3 4 5 6 7 8 9 10
```

The structure of the function is typical for recursive functions. They very often follow the following pattern.

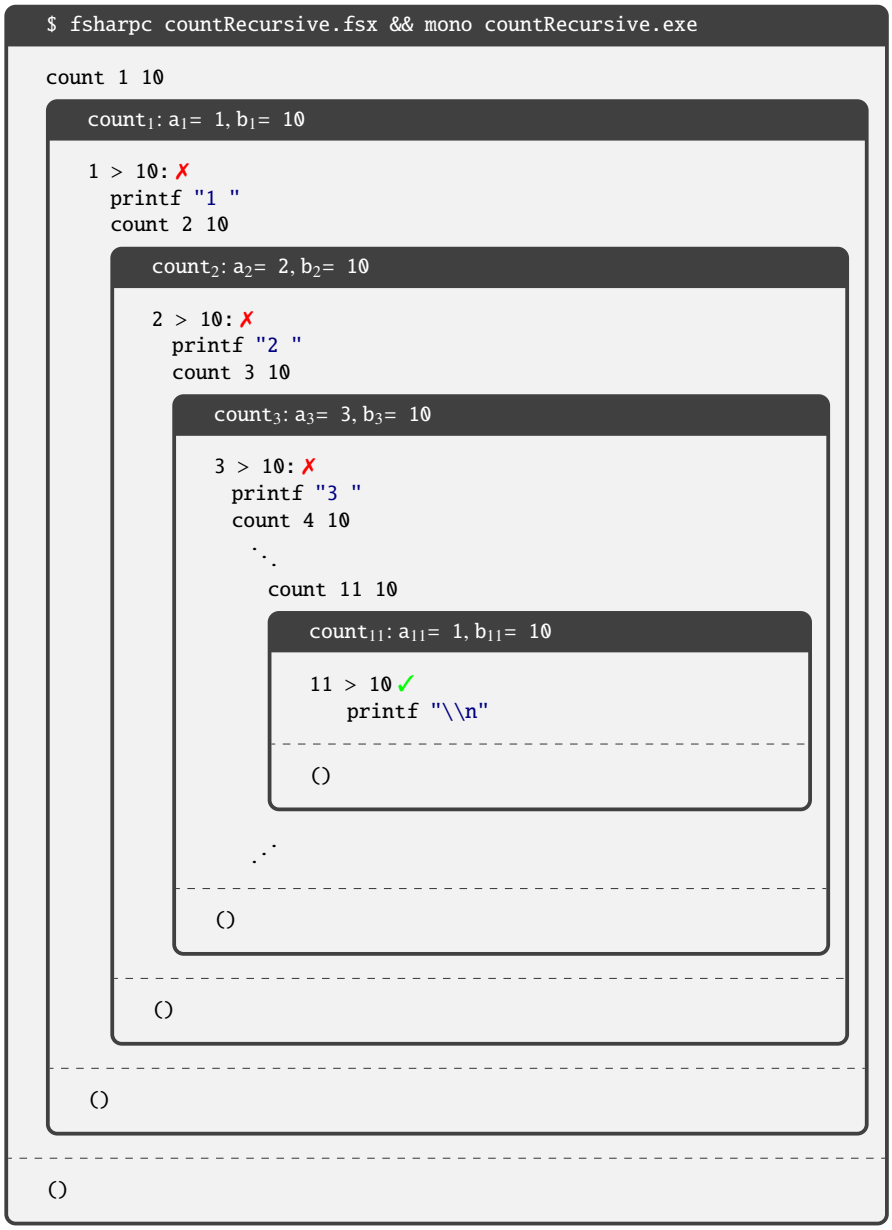


Fig. 8.2 Illustration of the recursion used to write the sequence "1 2 3 ... 10" in line 9 in Listing 8.2. Each frame corresponds to a call to `count`, where new values overshadow old ones. All calls return `unit`.

Listing 8.3: Recursive functions consist of a stopping criterium, a stopping expression, and a recursive step.

```
let rec f a =
  match a with
    <stopping pattern> ->
      <stopping step>
    | _ ->
      <recursion step>
```

The `if–then` is also a very common conditional structures. In Listing 8.2, `a > b` is the *stopping condition*, `printfn ""` is the *stopping step*, and `printfn "\%d " a; count (a + 1) b` is the *recursion step*.

A trick to designing recursive algorithms is to **assume that the recursive function already works**. For example, ⋆

```
let rec sum n = match n with 0 -> 0 | _ -> n + sum (n-1)
```

we call `sum` in the recursive step under the assumption, that it correctly sums up values from 0 to $n-1$. Thus, we are free only to consider how to calculate the sum from 0 to n given that we have n and the result of `sum (n-1)`.

Reconsider the Divide-by-two algorithm in Section 3.7. The algorithm consists of two elements. Given an initial unsigned integer `n`:

1. calculate `n%2u`. This will be the right-most digit in the binary representation of `n`,
2. solve the same problem but now for `n/2u` until `n=0`.

Hence, we have divided the total problem into a series of simple and identical steps. Following the design steps from Section 6.1, we decide to call the function `divideByTwo`, and define it as a mapping from unsigned integers to strings, `divideByTwo (n: uint): string`. From the above, we identify `n=0` as the base case, `divideByTwo (n/2u)` as the recursive call, and `divideByTwo (n/2u) + string (n%2u)` as the recursive step. The resulting algorithm can in brief be written as,

8.2 The Call Stack and Tail Recursion

Recursion is a powerful tool for designing compact and versatile algorithms. However, they can be slow and memory intensive. To understand this, we must understand

Listing 8.4 divideByTwoRecursive.fsx:
Implementing the divide-by-two algorithm using recursion.

```
let rec divideByTwo (n: uint) : string =
  match n with
    0u -> ""
    | _ -> (divideByTwo (n/2u)) + (string (n%2u))

printfn "13u_10 = %A_2" (divideByTwo 13u)
```

```
$ dotnet fsi divideByTwoRecursive.fsx
13u_10 = "1101"_2
```

how function calls are implemented in a typical language using the Call stack, and what can be done to make recursive functions efficient.

Consider Fibonacci's sequence of numbers which is a recursive sequence of numbers with relations to the Golden ratio and structures in biology. The Fibonacci sequence is the sequence of numbers $1, 1, 2, 3, 5, 8, 13, \ldots$. The sequence starts with $1, 1$ and the next number is recursively given as the sum of the two previous ones. A direct implementation of this is given in Listing 8.5. Here we extended the sequence to

Listing 8.5 fibRecursive.fsx:
The n'th Fibonacci number using recursion.

```
let rec fib (n: uint) =
  if n < 2u then n
  else fib (n - 1u) + fib (n - 2u)

printfn "%A" (List.map fib [0u .. 10u])
```

```
$ dotnet fsi fibRecursive.fsx
[0u; 1u; 1u; 2u; 3u; 5u; 8u; 13u; 21u; 34u; 55u]
```

$0, 1, 1, 2, 3, 5, \ldots$ with the starting sequence $0, 1$, allowing us to define a function for all non-negative integers without breaking the definition of the sequence. This ★ is a general piece of advice: **make functions which give sensible output for any argument from its input domain.**

A visualization of the calls and the scopes created by `fibRecursive` is shown in Figure 8.3. The figure illustrates that each recursive step results in two calls to the function, thus creating two new scopes. And it gets worse. Figure 8.4 illustrates the tree of calls for `fib 5`. Thus, a call to the function `fib` generates a tree of calls that is five levels deep and has `fib(5)` number of nodes. In general for the program in Listing 8.5, a call to `fib(n)` produces a tree with `fib(n)` $\leq c\alpha^n$ calls to the function for some positive constant c and $\alpha \geq \frac{1+\sqrt{5}}{2} \sim 1.6$. Each call takes time and requires memory, and we have thus created a slow and somewhat memory-intensive function.

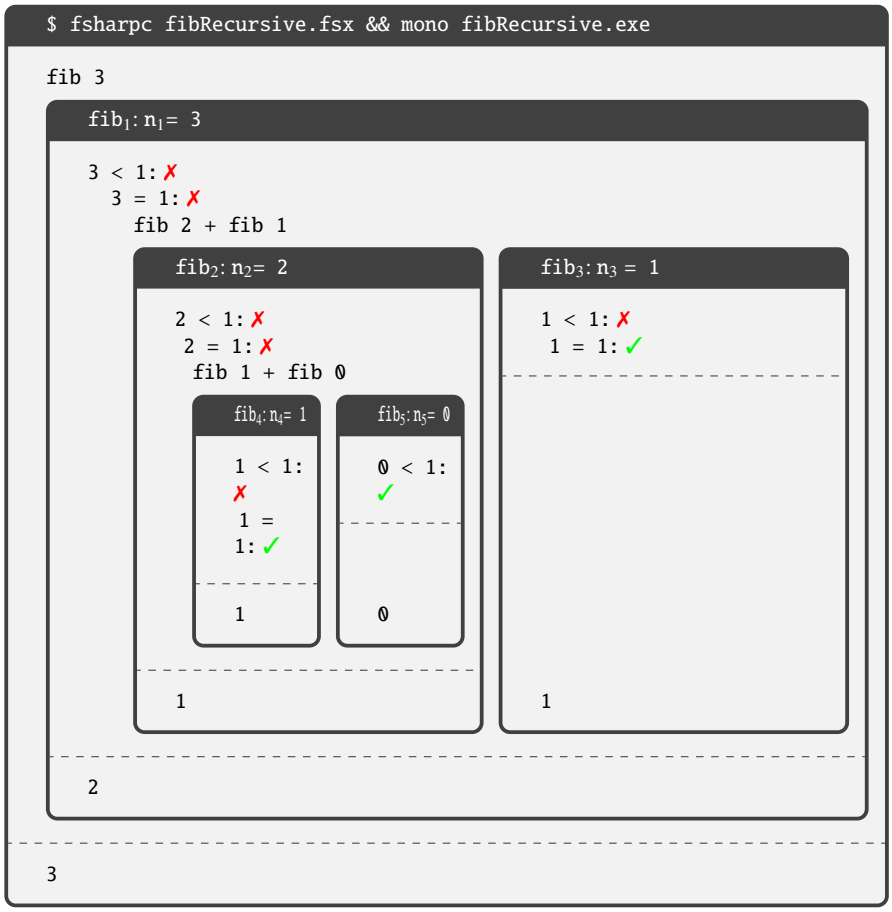


Fig. 8.3 Illustration of the recursion used to write the sequence "1 2 3 ... 10" in line 9 in Listing 8.2. Each frame corresponds to a call to `fib`, where new values overshadow old ones.

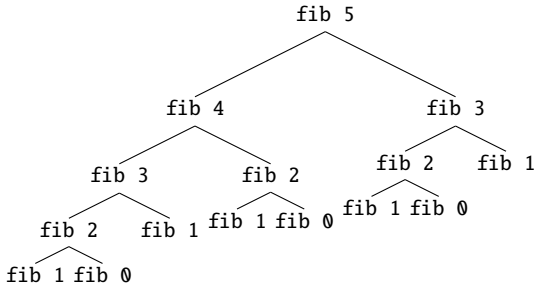


Fig. 8.4 The function calls involved in calling `fib 5`.

This is a hugely ineffective implementation of calculating entries into Fibonacci's sequence since many of the calls are identical. E.g., in Figure 8.4, `fib 1` is called five times. Before we examine a faster algorithm, we first need to discuss how F# executes function calls.

When a function is called, then memory is dynamically allocated internally for the function on what is known as the *call stack*. Stacks are used for many things in programming, but typically the call stack is considered special since it is almost always implicitly part of any program execution. Hence, it is often just referred to as *The Stack*. When a function is called, a new *stack frame* is stacked (pushed) on the call stack, including its arguments, local storage such as mutable values, and where execution should return to when the function is finished. When the function finishes, the stack frame is unstacked (popped) and in its stead, the return value of the function is stacked. This return value is then unstacked and used by the caller. After unstacking the return value, the call stack is identical to its state prior to the call. Figure 8.5 shows snapshots of the call stack when calling `fib 5` in Listing 8.5. The call first stacks a frame onto the call stack with everything needed to execute the

Fig. 8.5 A call to `fib 5` in Listing 8.5 starts a sequence of function calls and stack frames on the call stack.

function body plus a reference to where the return to, when the execution is finished. Then the body of `fib` is executed, which includes calling `fib 4` and `fib 3` in turn. The call to `fib 4` stacks a frame onto the call stack, and its body is executed. Once execution is returned from the call to `fib 4`, the result of the function is on top of the stack. It is unstacked, saved and the call to `fib 3` is treated equally. When the end of `fib 5` is reached, its frame is unstacked, and its result is stacked. In this way, the call stack is returned to its original state except for the result of the function, and execution is returned to the point right after the original call to `fib 5`. Thus, for Listing 8.5 $O\left(\alpha^n\right),\ \alpha = \frac{1+\sqrt{5}}{2}$ stacking operations are performed for a call to `fib n`. The $O\left(f(n)\right)$ is the *Landau symbol* used to denote the order of a function, such that if $g(n) = O(f(n))$ then there exists two real numbers $M > 0$ and a n_0 such that for all $n \geq n_0$, $|g(n)| \leq M|f(n)|$. As indicated by the tree in Figure 8.4, the call tree is at most n high, which corresponds to a maximum of n additional stack frames as compared to the starting point.

The implementation of Fibonacci's sequence in Listing 8.5 can be improved to run faster and use less memory. One such algorithm is given in Listing 8.6 Calculating the 45th Fibonacci number a MacBook Pro, with a 2.9 GHz Intel Core i5 using Listing 8.5 takes about 11.2s while using Listing 8.6 is about 224 times faster and only takes 0.050s. The reason is that `fib` in Listing 8.6 calculates every number in

Listing 8.6 fibRecursiveAlt.fsx:
A fast, recursive implementation of Fibonacci's numbers. Compare with Listing 8.5.

```
let fib (n: uint) =
  let rec fibPair n pair =
    if n < 2u then pair
    else fibPair (n - 1u) (snd pair, fst pair + snd pair)

  if n < 2u then n
  else fibPair n (0u, 1u) |> snd

printfn "fib(10) = %A" (fib 10u)
```

```
$ dotnet fsi fibRecursiveAlt.fsx
fib(10) = 55u
```

the sequence once and only once by processing the list recursively while maintaining the previous two values needed to calculate the next in the sequence. I.e., the function `fibPair` transforms the pair `(a,b)` to `(b,a+b)` such that, e.g., the 4th and 5th pair `(3,5)` is transformed into the 5th and the 6th pair `(5,8)` in the sequence. What complicates the algorithm is that besides the transformation, we must keep track of when to stop, which here is done using a counter variable, that is recursively reduced by 1 until our stopping criterium.

Listing 8.6 also uses much less memory than Listing 8.5, since its recursive call is the last expression in the function, and since the return value of two recursive calls to `fibPair` is the same as the return value of the last. In fact, the return value of any number of recursive calls to `fibPair` is the return value of the last. This structure is called *tail-recursion*. Compilers can easily optimize the call stack usage for tail recursion, since when in this example `fibPair` calls itself, then its frame is no longer needed, and may be replaced by the new `fibPair` with the slight modification, that the return point should be to `fib` and not the end of the previous `fibPair`. Once the recursion reaches the stopping criteria, then instead of popping a long list of calls of `fibPair` frames, then there is only one, and the return value is equal to the return value of the last call and the return point is to `fib`. Thus, many stack frames in tail recursion are replaced by one. Hence, **prefer tail-recursion whenever possible.** ⋆

8.3 Mutually Recursive Functions

Functions that recursively call each other are called *mutually recursive* functions. F# offers the *let – rec – and* notation for co-defining mutually recursive functions. As an example, consider the function `even : int -> bool`,

which returns true if its argument is even and false otherwise, and the opposite function `odd : int -> bool`. A mutually recursive implementation of these functions can be developed from the following relations: `even 0 = true`, `odd 0 = false`, and for $n > 0$, `even n = odd (n-1)`, which implies that for $n > 0$, `odd n = even (n-1)`: Notice that in the lightweight notation the `and` must

Listing 8.7 mutuallyRecursive.fsx:
Using mutual recursion to implement even and odd functions.

```
let rec even x =
  if x = 0 then true
  else odd (x - 1)
and odd x =
  if x = 0 then false
  else even (x - 1)

let res = List.map (fun i -> (i, even i, odd i)) [1..3]
printfn "(i, even, odd):\n%A" res
```

```
$ dotnet fsi mutuallyRecursive.fsx
(i, even, odd):
[(1, false, true); (2, true, false); (3, false, true)]
```

be on the same indentation level as the original `let`. Without the `and` keyword, F# will issue a compile error at the definition of `even`.

In the example above, we used the `even` and `odd` function problems to demonstrate mutual recursion. There is, of course, a much simpler solution, which does not use recursion at all: A better way to test for parity without recursion. This is to be

Listing 8.8 parity.fsx:
parity

```
let even x = (x % 2 = 0)
let odd x = not (even x)
let res = List.map (fun i -> (i, even i, odd i)) [1..3]
printfn "(i, even, odd):\n%A" res
```

```
$ dotnet fsi parity.fsx
(i, even, odd):
[(1, false, true); (2, true, false); (3, false, true)]
```

preferred anytime as the solution to the problem.

8.4 Recursive types

Data types can themselves be recursive. As an example, the linked list in Figure 7.2. Pattern matching must be used in order to define functions on values of a discrimi-

Listing 8.9 discriminatedUnionList.fsx:
A discriminated union modelling for linked lists.

```
type Lst = Ground | Element of int*Lst

let lst = Element (1, Element (2, Element (3, Ground)))
printfn "%A" lst
```

```
$ dotnet fsi discriminatedUnionList.fsx
Element (1, Element (2, Element (3, Ground)))
```

nated union. E.g., in Listing 8.10 we define a function that traverses a list and prints the content of the elements. Discriminated unions are very powerful and can often

Listing 8.10 discriminatedUnionPatternMatching.fsx:
Traversing a recursive list type with pattern matching.

```
type Lst = Ground | Element of int*Lst
let rec traverse (l : Lst) : string =
    match l with
      Ground -> ""
      | Element(i,Ground) -> string i
      | Element(i,rst) -> string i + ", " + (traverse rst)

let lst = Element (1, Element (2, Element (3, Ground)))
printfn "%A" (traverse lst)
```

```
$ dotnet fsi discriminatedUnionPatternMatching.fsx
"1, 2, 3"
```

be used instead of class hierarchies. Class hierarchies are discussed in Section 16.1.

8.5 Tracing Recursive Programs

Tracing by hand is a very illustrative method for understanding recursive programs. Consider the recursive program in Listing 8.11. The program includes a function for calculating the greatest common divisor of 2 integers, and calls this function with the numbers 10 and 15. Following the notation introduced in Section 4.5, we write:

Listing 8.11 gcd.fsx:
The greatest common divisor of 2 integers.

```
1  let rec gcd a b =
2    if a < b then
3      gcd b a
4    elif b > 0 then
5      gcd b (a % b)
6    else
7      a
8
9  let a = 10
10 let b = 15
11 printfn "gcd %d %d = %d" a b (gcd a b)
```

```
1 $ fsharpc --nologo gcd.fsx && mono gcd.exe
2 gcd 10 15 = 5
```

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	gcd = $\big((a, b)$, gcd-body, $()\big)$
2	9	E_0	$a = 10$
3	10	E_0	$b = 15$

In line 11, `gdc` is called before any output is generated, which initiates a new environment E_1 and executes the code in gcd-body:

Step	Line	Env.	Bindings and evaluations
4	11	E_0	gcd a b = ?
5	1	E_1	$\big((a = 10, b = 15)$, gcd-body, $()\big)$

In E_1 we have that $a < b$, which fulfills the first condition in line 2. Hence, we call gdc with switched arguments and once again initiate a new environment,

Step	Line	Env.	Bindings and evaluations
6	2	E_1	a<b = true
7	3	E_1	gcd b a = ?
8	1	E_2	$\big((a = 15, b = 10)$, gcd-body, $()\big)$

In E_2, `a < b` in line 2 is false, but `b > 0` in line 4 is true, hence, we first evaluate `a % b`, call `gcd b (a % b)`, and then create a new environment,

Step	Line	Env.	Bindings and evaluations
9	2	E_2	a<b = false
10	4	E_2	b>0 = true
11	5	E_2	a % b = 5
12	5	E_2	gcd b (a % b) = ?
13	1	E_3	$\big((a = 10, b = 5)$, gcd-body, $()\big)$

Again we fall through to line 5, evaluate the remainder operator, and initiate a new environment,

Step	Line	Env.	Bindings and evaluations
14	2	E_3	a<b = false
15	4	E_3	b>0 = true
16	5	E_3	a % b = 0
17	5	E_3	gcd b (a % b) = ?
18	1	E_4	$\big((a = 5, b = 0)$, gcd-body, $()\big)$

This time both `a < b` and `b > 0` are false, so we fall through to line 7 and return the value of `a` from E_4, which is `5`:

Step	Line	Env.	Bindings and evaluations
19	2	E_4	a<b = false
20	4	E_4	b>0 = false
21	7	E_4	return = 5

We scratch E_4, return to E_3, replace the ?-mark with `5`, and continue the evaluation of line 5. Since this is also a branch of the last statement in `gdc`, we return the previously evaluated value,

Step	Line	Env.	Bindings and evaluations
22	5	E_3	gcd b (a % b) = 5
23	5	E_3	return = 5

Like before, we scratch E_3, return to E_2, replace the ?-mark with `5`, and continue the evaluation of line 5. Since this is also a branch of the last statement in `gdc`, we return the just evaluated value,

Step	Line	Env.	Bindings and evaluations
24	5	E_2	gcd b (a % b) = 5
25	5	E_2	return = 5

Again, we scratch E_2, return to E_1, replace the ?-mark with `5`, and continue the evaluation of line 5. Since this is also a branch of the last statement in `gdc`, we return the just evaluated value,

Step	Line	Env.	Bindings and evaluations
26	3	E_1	gcd a b = 5
27	3	E_1	return = 5

Finally, we scratch E_1, return to E_0, replace the ?-mark with `5`, and continue the evaluation of line 11:

Step	Line	Env.	Bindings and evaluations
28	11	E_0	gcd a b = 5
29	11	E_0	output = "gcd a b = 5"
30	11	E_0	return = ()

Note that the output of `printfn` is a side-effect while its return-value is unit. In any case, since this is the last line in our program, we are done tracing.

8.6 Key Concepts and Terms in This Chapter

In this chapter, we have taken a second look at recursion. You have seen:

- how to define *recursive functions* and *mutually recursive functions*.
- how the *call stack* influences the resources used by recursive functions, and how *tail recursion* is a method for making recursive function efficient.
- *recursive discriminated unions* and how they can be used to model linked lists.
- how to trace-by-hand recursive functions.

Chapter 9

Organising Code in Libraries and Application Programs

Abstract We have previously seen how code can be organized into functions to make programs easier to read, make code pieces reusable, and make programs easier to debug. Functions and values may further be grouped into libraries, and the `List` module is an example of such a library that you have already used. F# includes several programming structures to organize code in libraries: Modules, namespaces, and classes. In this chapter, we will focus on modules. Classes will be described in detail in Chapter 15. Here you will learn how to:

- Use the `dotnet` command-line tool to create project files and how to compile these into an executable file.
- The difference between running interpreted and compiled programs.
- Make libraries using modules and write applications using such libraries.
- Specify the abstract structure of a module using signature files.
- Create an implementation of the Stack abstract datatype.
- Update an integer stack to a generic stack.

9.1 Dotnet projects: Libraries and applications

As our programs grow in size, it can be convenient to split the program over several files, e.g., by separating functionality into something general and specific for the problem being solved. An example of this is the `List` module which contains general functions on lists, and which you have used in your programs. In this chapter, we will write modules ourselves, also known as libraries, and the programs using these libraries, we will call applications. Using the `dotnet` command-line tool, we are able to create project files which have a `.fsproj` suffix, which include information about which source code and packages belongs together. The `dotnet` command-line tool can help structure the files on the filesystem by use of directories, but here we advocate for a simple, hand-held solution.

To make a light version of `dotnet` project with a library and an application file, start by creating the `dotnet` project template as follows:

Listing 9.1: Creating an initial library-application file setup.

```
$ dotnet new console -lang "F#" -o app
```

This creates the `app` directory with among other things a `Program.fs` file. The `Program.fs` is the default filenames for an application. Usually, the `.fs` suffix is reserved for libraries, so, rename `Program.fs` to `Program.fsx`. Then create a possibly empty library file `Library.fs` using a standard editor. You should now have a directory as shown in Figure 9.1. The `.fsproj` file is an XML-file which

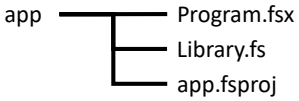


Fig. 9.1 A set of files for a light version of a dotnet project.

describes how dotnet should combine various files. The `dotnet` command-line tool can edit this file, but we might as well do this ourselves in a text editor: Open the file in your favorite text editor and in the `<ItemGroup>` change `Program.fs` to `Program.fsx` and add a line with the name of your library file `<Compile Include="Library.fs" />`. The resulting file should look like this:

Listing 9.2: The initial content of `app.fsproj`.

```
<Project Sdk="Microsoft.NET.Sdk">
 <PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net6.0</TargetFramework>
 </PropertyGroup>
 <ItemGroup>
  <Compile Include="Library.fs" />
  <Compile Include="Program.fsx" />
 </ItemGroup>
</Project>
```

If you decide to rename the application or library files, then you must update the project files accordingly.

If you wish to add references to packages such as the `DIKU.Canvas` package, this can be done as,

Listing 9.3: Creating an initial library-application file setup.

```
$ dotnet add app/app.fsproj package "DIKU.Canvas" --version
   1.0.1
```

or manually by editing the project file appropriately. When a package is included in the project file, then it does not need to be loaded in libraries and applications using the `#r` directive. This version will compile and run the library and the program, but will not build the library separately.

The order of the references to packages, libraries, and application files are important, since `dotnet` will read them from top to bottom, and only if, e.g., `Library.fs` is above `Program.fsx` will the library functions be available in the application.

As an example, change `Program.fs` to become what is shown in Listing 9.4, change

Listing 9.4 solution/app/Program.fs:
A simple application program.

```
open Library

printfn "%A" (greetings "Jon")
```

`Library.fs` to become what is shown in Listing 9.5, and run it in *compile mode* by

Listing 9.5 solution/library/Library.fs:
A simple library.

```
module Library

let greetings (str: string) : string = "Greetings " + str
```

changing to the `app` directory and using the `dotnet run` command as demonstrated in Listing 9.6.

Listing 9.6: Running an application setup with one or more project files.

```
$ cd solution/app
$ dotnet run
"Greetings Jon"
```

Assuming that `Program.fs` was renamed to `Program.fsx` and `app.fsproj` was edited appropriately, `dotnet run` is almost the same as

Listing 9.7: Running an application setup with one or more project files.

```
$ dotnet fsi ../library/Library.fs Program.fsx
"Greetings Jon"
```

However, `dotnet fsi` *interprets* the library and application into executable code everytime it is called, while `dotnet run` only *compiles* the program once. On my laptop, the time these different steps take depends on what else is running on the computer, but typical timings are

Command	Time
`dotnet fsi ../library/Library.fs Program.fsx`	1.2s
`dotnet run` (first time)	4.0s
`dotnet run`	1.0s

The example application, we are studying here, is tiny, but even in this case, the repeated translation by `dotnet fsi` is a 16% overhead when compared to an already compiled program, and you should expect this overhead to be larger for larger programs. However, for tiny programs, the cost of the initial compilation is 400% and not worth the effort from a time perspective.

9.2 Libraries and applications

A library in F# is expressed as a *module*, which is a programming structure used to organize type declarations, values, functions, etc. The libraries should have the suffix `.fs`, and here will call them *implementation files* in contrast to the signature files to be discussed below, which we will call *signature files*.

A module is typically a file where the module name is declared in the first lines using the `module` with the following syntax,

Listing 9.8: Outer module.

```
module <ident>
<script>
```

Here, the identifier `<ident>` is a name not necessarily related to the filename, and the script `<script>` is an expression.

Consider the example from Listing 6.4 in which functions are defined for solving the values of x where $f(x) = 0$ for a quadratic equation. In the following, we will split this into a library of functions and an application program. For this, we set up a project system of files as described in Section 9.1, where `Program.fs` has been replaced by `Program.fsx` and the `app.fsproj` has been edited appropriately. The content of `Library.fs` has been changed to become what is shown in Listing 9.9, and `Program.fsx` has been changed to what is shown in Listing 9.10.

Listing 9.9 solve/library/Library.fs:
A library for solving quadratic equations.

```
module Solve

let discriminant a b c = b ** 2.0 - 4.0 * a * c

let solveQuadraticEquation a b c =
  let d = discriminant a b c
  ((-b + sqrt d) / (2.0 * a),
   (-b - sqrt d) / (2.0 * a))
```

Listing 9.10 solve/app/Program.fsx:
An application using the `Solve` module.

```
open Solve

let p1 = solveQuadraticEquation 1.0 0.3 -1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p1
let p2 = solveQuadraticEquation 1.0 0.0 0.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p2
let p3 = solveQuadraticEquation 1.0 0.0 1.0
printfn "0=1.0x^2+0.3x-1.0 => x = %A" p3
```

9.3 Specifying a Module's Interface with a Signature File

As the 8-step guide suggests, the design of programs is helped by first considering what the program's function is to do before actually implementing them. This also holds for libraries, and signature files can aid this process.

A *signature files* is a file accompanying *implementation files* and have the suffix `.fsi`. A signature file contains almost no implementation, but only type definitions. Signature files offer three distinct features:

1. Signature files can be used as part of the documentation of code since type information is of paramount importance for an application programmer to use a library.

2. Signature files may be written before the implementation file. This allows for a higher-level programming design that focuses on *which* functions should be included and *how* they can be composed.

3. Signature files allow for access control. Most importantly, if a type definition is not available in the signature file, then it is not available to the application program. Such definitions are private and can only be used internally in the library code. More fine-grained control related to classes is available and will be discussed in Chapter 15.

These features help the programmer structure the process of programming and protect the user of a library from irrelevant data and functions. A signature file contains the type definitions and the types of values and functions to be exposed to the user of the library. For example, for the library in Listing 9.9, we can define a signature files which makes the `solveQuadraticEquation` function but not the `discriminant` function available to the user of the library as demonstrated in Listing 9.11. To compile the application using the signature file, we must add the

Listing 9.11 solve/library/Library.fsi: A signature file for Listing 9.9.

```
1 module Solve
2
3 val solveQuadraticEquation: a: float -> b: float -> c: float
     -> float*float
```

created file, e.g., `Library.fsi`, to the project file as, e.g., shown in Listing 9.12.

Listing 9.12: The `library.fsproj` with a signature file added.

```
<Project Sdk="Microsoft.NET.Sdk">
 <PropertyGroup>
  <OutputType>Exe</OutputType>
  <TargetFramework>net6.0</TargetFramework>
 </PropertyGroup>
 <ItemGroup>
  <PackageReference Include="DIKU.Canvas" Version="1.0.1" />
  <Compile Include="Library.fsi" />
  <Compile Include="Library.fs" />
  <Compile Include="Program.fsx" />
 </ItemGroup>
</Project>
```

In the context of the 8-step guide, it is useful to write the signature file before the implementation file, and that the signature file contains the documentation for the functions available in the application.

For technical reasons in the `dotnet` framework, exposed type abbreviations must be given both in the signature file and the implementation file. The implication is that the signature at times must also define implementation details, see e.g., the example below, and thus becomes less abstract the desirable in general.

9.4 Programming Intermezzo: Postfix Arithmetic with a Stack

To this point, we have performed simple arithmetic using *infix* notation, meaning that expressions like $(4+6*3)/2-8$ are evaluated using the precedence and association rules of the operators as

$$\begin{aligned}(4+6*3)/2-8 &\rightsquigarrow (4+18)/2-8 && (9.1)\\ &\rightsquigarrow 22/2-8 && (9.2)\\ &\rightsquigarrow 11-8 && (9.3)\\ &\rightsquigarrow 3 && (9.4)\end{aligned}$$

However, there is an equally valid notation, *postfix*, in which the same expression is written as 4 6 3 $*$ + 2 / 8 −. Here, the rule is to read from left to right, and whenever there are two values and an operator, a b op, replaced this with the value a op b and repeat until only one value remains, which is the result of the calculation. Hence,

$$4\ 6\ 3\ *\ +\ 2\ /\ 8\ - \rightsquigarrow 4\ 18\ +\ 2\ /\ 8\ - \tag{9.5}$$
$$\rightsquigarrow 22\ 2\ /\ 8\ - \tag{9.6}$$
$$\rightsquigarrow 11\ 8\ - \tag{9.7}$$
$$\rightsquigarrow 3 \tag{9.8}$$

This was implemented on a series of calculators released by Hewlett-Packard in the 1960-1980'ies, and one of the arguments for this notation was, that the expressions could be evaluated by a stack with only 3 levels. In the following, we will look at stacks as an abstract datatype and build a library for stacks and an arithmetic solver for such simple expressions using this stack.

A *stack* is an abstract datatype, meaning that it is defined by its concepts, not its implementation. The concept of a stack is like a stack of plates in a cafeteria, they are placed in a physical stack, and you can take the top plate and place a plate on the top, but you cannot access a plate in the middle of a stack. Stacks typically come with the following functions:

create: Create an empty stack.

pop: Return the top element and the resulting stack.

push: Put an element on a stack and return the resulting stack.

isEmpty: Check whether the stack is empty.

Following the 8-step guide Section 6.1, the above directly suggests names and includes brief descriptions (Steps 1 and 2). Step 3 suggests that we write a simple test, and since we are fond of piping, our test program is shown in Listing 9.13. We

Listing 9.13 postfixTest.fsx:
A simple program using a yet to be written library.

```
1 open Stack
2
3 create () |> push 1.0 |> push 2.0 |> pop |> printfn "%A"
```

expect this to print the result of the last `pop` call, which should include information about the element `2`.

In the functional programming paradigm, our stack is a constant, implying that every time we pop and push, we create new stacks. Thus, for step 4 in the 8-step guide, we must accept that all but `isEmpty` returns a new stack, and all but `create` must take a stack as input. Thus we arrive at a signature file for the stack-library given in Listing 9.14. A limitation to F#'s modules is that the type specifications need explicit declaration. We would have liked to write `type stack` and functions of some variable type `'e`, since the stack concept is independent of the type of elements

Listing 9.14 postfixLibrary.fsi:
A signature file for the stack library

```
module Stack

type stack = float list // a stack of elements

// create an empty stack
val create: unit -> stack
// return the top element and the resulting stack
val pop: stack -> float * stack
// put an element on a stack and return the resulting stack
val push: float -> stack -> stack
// check whether the stack is empty
val isEmpty: stack -> bool
```

it contains. However, this is not possible, and thus, we here specialize to float stacks. For similar reasons, we are forced to specify details about the implementation of our type. Our idea is that stacks can be implemented as lists since lists are well suited to work with the first elements.

Implementing a stack using lists is simple, since lists already contains the properties `Head`, `Tail`, and `IsEmpty`, which closely mimics the needed operations for a stack. Thus we arrive at Listing 9.15. And now we can run our test code as shown in

Listing 9.15 postfixLibrary.fs:
An implementation of a stack module.

```
module Stack

type stack = float list
let create () : stack = []
let pop (stck: stack) : float*stack =  (stck.Head, stck.Tail)
let push (elm: float) (stck: stack): stack = elm::stck
let isEmpty (stck: stack): bool = stck.IsEmpty
```

Listing 9.16. As expected, the top element and the resulting stack is `(2,[1])`.

Listing 9.16: Running the test program

```
$ dotnet fsi postfixLibrary.fsi postfixLibrary.fs
   postfixTest.fsx
(2.0, [1.0])
```

To implement simple postfix algebra, we will use discriminated unions. That is, we define a type,

```
type element = Value of int | Multiply | Plus | Minus | Divide
```

This allows us to make lists of tokens such as,

```
[Value 4; Value 6; Value 3; Multiply; Plus]
```

for the expression 3 4 2 / + which is equivalent to $3+4/2$ in infix notation. The next step is to understand how to use a stack to evaluate such expressions. The idea is to process the list of tokens from its head and push values to a stack. When the head of the tokens list is an operator, say 'op' then the two top elements from the stack are popped, say a and b, the mathematical expression $c = a$ op b is evaluated and c is pushed to the stack. For our example, the evolution of the stack will be:

Unused tokens	Evaluation stack
4 6 3 ∗ + 2 / 8 −	[]
6 3 ∗ + 2 / 8 −	[4]
3 ∗ + 2 / 8 −	[6; 4]
∗ + 2 / 8 −	[3; 6; 4]
+ 2 / 8 −	[18; 4]
2 / 8 −	[22]
/ 8 −	[2; 22]
8 −	[11]
−	[8; 11]
	[3]

As demonstrated, the result of the expression is the last element on the stack, once the list of tokens is empty. An F# implementaiton is given in Listing 9.17.

9.5 Generic Modules

The stack is an example of an abstract datatype, and in the previous section, we implemented a stack for floats, however, stacks can be of many other types, and although we could make a stack module for each type, it would greatly improve the usefulness of our library, if we could make a stack module, which is generic, i.e., where the user can decide when writing applications, which type of values to stack. Luckily, this is supported in F#.

In Section 5.5 we discussed the usefulness of the variable type such as `'a`, which makes functions and types generic, that is, the same definition can be used for any type. To make a generic module, it is often useful first to make a non-generic version, such as our float stack, since it is often easier to spot errors in concrete program versions. The float stack already works as desired, so we will now modify the module to be a stack for a variable type. In our example, we must update both the type abbreviation and function types in the signature file. The result is shown in Listing 9.18. The next step is to update the implementation file. During such

Listing 9.17 postfixApp.fsx:
An application for evaluating lists of tokens on postfix form using a stack.

```
open Stack

type element = Value of float | Mul | Add | Sub | Div

let tokens = [Value 4.0; Value 6.0; Value 3.0; Mul; Add;
   Value 2.0; Div; Value 8.0; Sub]

let rec eval (tkns: element list) (stck: stack): stack =
  match tkns with
    [] -> stck
    | elm::rst ->
      match elm with
        Value v ->
          push v stck |> eval rst
        | Mul ->
          let (a, stck1) = pop stck
          let (b, stck2) = pop stck1
          push (b*a) stck2 |> eval rst
        | Add ->
          let (a, stck1) = pop stck
          let (b, stck2) = pop stck1
          push (b+a) stck2 |> eval rst
        | Sub ->
          let (a, stck1) = pop stck
          let (b, stck2) = pop stck1
          push (b-a) stck2 |> eval rst
        | Div ->
          let (a, stck1) = pop stck
          let (b, stck2) = pop stck1
          push (b/a) stck2 |> eval rst

printfn "%A = %A" tokens (eval tokens (create ()))
```

```
$ dotnet fsi postfixLibrary.fsi postfixLibrary.fs
   postfixApp.fsx
[Value 4.0; Value 6.0; Value 3.0; Mul; Add; Value 2.0; Div;
   Value 8.0; Sub] = [3.0]
```

transformations, it is not uncommon to realize that restrictions must be put on the type, which is possible but which we will not consider further in this book. Since the stack does not rely on any properties of the stack elements, there is no challenge in modifying the implementation file as shown in Listing 9.19. Finally, we can make an application using stacks of various kinds, as shown in Listing 9.20. In the program, we see that F# can infer that a stack, on which floats are pushed, must be of `stack<float>` type, and when characters are pushed, then the stack must be of type `stack<char>`. Thus, we have arrived at a stack for any type, whose interface

Listing 9.18 postfixLibraryGeneric.fsi:
A signature file for the generic stack library

```
module Stack

type stack<'a> = 'a list // a stack of elements

// create an empty stack
val create: unit -> stack<'a>
// return the top element and the resulting stack
val pop: stack<'a> -> 'a * stack<'a>
// put an element on a stack and return the resulting stack
val push: 'a -> stack<'a> -> stack<'a>
// check whether the stack is empty
val isEmpty: stack<'a> -> bool
```

Listing 9.19 postfixLibraryGeneric.fs:
An implementation of a generic stack module.

```
module Stack

type stack<'a> = 'a list
let create () : stack<'a> = []
let pop (stck: stack<'a>) : 'a*stack<'a> =  (stck.Head,
   stck.Tail)
let push (elm: 'a) (stck: stack<'a>): stack<'a> = elm::stck
let isEmpty (stck: stack<'a>): bool = stck.IsEmpty
```

Listing 9.20 postfixTestGeneric.fsx:
Running the test program

```
open Stack

create () |> push 1.0 |> push 2.0 |> pop |> printfn "%A"
create () |> push 'a' |> push 'b' |> pop |> printfn "%A"
```

```
$ dotnet fsi postfixLibraryGeneric.fsi
   postfixLibraryGeneric.fs postfixTestGeneric.fsx
(2.0, [1.0])
('b', ['a'])
```

is given by the signature file, and almost all of its implementation is hidden in the implementation file.

9.6 Key Concepts and Terms in This Chapter

In this chapter, we have looked at how to build libraries and applications. Key concepts have been

- How to build organise the compilation of libraries and applications using `dotnet` **project files**.
- How to design libraries using **modules** and **signature files**.
- How to make **library implementation files**.
- How to implement the **abstract datatype Stack** both as a float stack and as a **generic module**.

Chapter 10

Higher-Order Functions

Abstract A *higher-order function* is a function that takes a function as an argument and/or returns a function. higher-order functions are also sometimes called functionals or functors. In this chapter you will learn how to:

- make functions that take and/or return functions as values.
- create new functions with the function composition operator.
- create new functions with a partial specification of function arguments.

10.1 Functions as Values

F# is a functions-first programming language with strong support for working with functions as values: Functions evaluate as *closures*, see Section 4.2, which can be passed to and from functions as any other value. An example of a higher-order function is `List.map` which takes a function and a list and produces a list, demonstrated in Listing 10.3. Here `List.map` applies the function `inc` to every element of the

Listing 10.1 higherOrderListMap.fsx:
`List.map` is a higher-order function, since it takes a function as argument.

```
let inc x = x + 1
let newList = List.map inc [2; 3; 5]
printfn "%A" newList
```

```
$ dotnet fsi higherOrderListMap.fsx
[3; 4; 6]
```

list. higher-order functions are often used together with *anonymous functions*, where the anonymous function is given as an argument. For example, Listing 10.3 may be rewritten using an anonymous function as shown in Listing 10.2.

Listing 10.2 higherOrderAnonymous.fsx:
An anonymous function is a higher-order function used here as an unnamed argument. Compare with Listing 10.3.

```
let newList = List.map (fun x -> x + 1) [2; 3; 5]
printfn "%A" newList
```

```
$ dotnet fsi higherOrderAnonymous.fsx
[3; 4; 6]
```

Writing a function that takes other functions as arguments is straightforward. If we were to make our own `map` function, it could look like what is shown in Listing 10.3. In this case, `map` has the type

```
map: f: ('a -> 'b) -> lst: 'a list -> 'b list
```

All three programs result in the same output and as such are equal. Likewise, running times will be equal. However, they differ in readability for a programmer and ease of bug hunting and future maintenance: Bindings allow us to reuse the code at a later stage, but if there is no reuse, then the additional bindings may result in a cluttered program. Nevertheless, programs using anonymous functions can be difficult to debug. Finally, bindings emphasize semantic aspects of the evaluation

Listing 10.3 higherOrderMap.fsx:
A homemade version of `List.map`.

```
let rec map f lst =
  match lst with
    [] -> []
    | e::rst -> (f e)::map f rst

let newList = map (fun x->x+1) [2; 3; 5]
printfn "%A" newList
```

```
$ dotnet fsi higherOrderMap.fsx
[3; 4; 6]
```

being performed merely by the names we select, and typically long, meaningful names are to be preferred, within reasonable limits. For example, instead of `inc` one could have used `increment_by_one` or similar which certainly is semantically meaningful, but many programmers will find that the short is to be preferred in order to reduce the amount of typing to be performed.

Functions can also be return values. For example, in Listing 10.4, the function `incBy` creates functions, which increments by a given argument. Note that the closure of

Listing 10.4 higherOrderReturn.fsx:
The procedure `inc` returns an increment function. Compare with Listing 10.3.

```
let incBy n =
  fun x -> x + n
printfn "%A" (List.map (incBy 2) [2; 3; 5])
```

```
$ dotnet fsi higherOrderReturn.fsx
[4; 5; 7]
```

this customized function is only produced once when the arguments for `List.map` are prepared, and not every time `List.map` applies the function to the elements of the list. Compare with Listing 10.3.

10.2 The Function Composition Operator

F# has strong support for working with functions on a functional level. In Section 4.2 on page 56, we saw how functions can be composed by passing the result of one function to the next, e.g., using piping. Alternatively, we can compose functions

before we apply them to values using the "`>>`" and "`<<`" *composition operators*, which is defined as,

```
(>>) : ('a -> 'b, 'b -> 'c) -> ('a -> 'c)
(<<) : ('b -> 'c, 'a -> 'b,) -> ('a -> 'c)
```

i.e., it takes two functions of type `'a -> 'b` and `'b -> 'c` respectively, and produces a new function of type `a' -> 'c`. As an example, consider the composition of the $\log x$ and $\sqrt{x}$ functions to make $f(x) = \log\left(\sqrt{x}\right), x > 0$. Using the piping operator, this can be written as `x |> sqrt |> log`, which is sufficient, if the function is only to be used once or not passed as an argument. However, the "`>>`" operator allows us to make a new function by `logSqrt = sqrt >> log` or equivalently `logSqrt = log << sqrt`. As with the piping operators, the precedence and association rules imply differences in the number of parentheses needed, but in the end, the choice mostly boils down to personal preference. In Listing 10.5 is a comparison of regular composition and the composition operator is shown.

Listing 10.5 functionPipingAdv.fsx:
A demonstration of differences in function composition.

```
let lst = [1.0..3.0]
// regular composition
printfn "%A" (List.map (fun x -> log (sqrt x)) lst)
// piping operator
printfn "%A" (List.map (fun x -> x |> sqrt |> log) lst)
printfn "%A" (List.map (fun x -> log <| (sqrt <| x)) lst)
// composition operator
printfn "%A" (List.map (sqrt>>log) lst)
printfn "%A" (List.map (log<<sqrt) lst)
```

```
$ dotnet fsi functionPipingAdv.fsx
[0.0; 0.3465735903; 0.5493061443]
[0.0; 0.3465735903; 0.5493061443]
[0.0; 0.3465735903; 0.5493061443]
[0.0; 0.3465735903; 0.5493061443]
[0.0; 0.3465735903; 0.5493061443]
```

10.3 Currying

Functions, with only the initial list of arguments, also return functions. This is called *partial specification* or *currying* in tribute of Haskell Curry[1]. An example is given in Listing 10.6. Here, `mul 2.0` is a partial application of the function `mul x y`,

Listing 10.6 higherOrderCurrying.fsx:
Currying: defining a function as a partial specification of another.

```
let mul x y = x*y
let timesTwo = mul 2.0
printfn "%g" (mul 5.0 3.0)
printfn "%g" (timesTwo 3.0)
```

```
$ fsharpc --nologo higherOrderCurrying.fsx && mono
   higherOrderCurrying.exe
15
6
```

where the first argument is fixed, and hence `timesTwo` is a function of 1 argument being the second argument of `mul`.

Currying is emphasized by how the type of functions of several values is written. Consider a function `f` of two generic arguments. Its type in F# will be `f : 'a -> 'b -> 'c`, meaning that `f` takes an argument of type `'a` and returns a function of type `'b -> 'c`. That is, if just one argument is given, then the result is a function, not a value. This is illustrated in Figure 10.1.

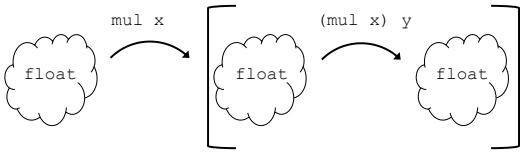


Fig. 10.1 The type of `mul x y = x*y` is a function of 2 arguments is a function from values to functions to values.

Currying makes elegant programs and is often used in functional programming. Nevertheless, currying may lead to obfuscation, and in general, **currying should be used with care and be well documented for proper readability of code.** ★

[1] Haskell Curry (1900–1982) was an American mathematician and logician who also has a programming language named after him: Haskell.

10.4 Key concepts and terms in this chapter

In this chapter, we have looked at higher-order functions. Key concepts have been:

- **Functions are values** and can be used as arguments and return value.
- Functions can be compose using the **composition operator** to produce new functions.
- Function arguments can be **partially specified** to create new functions with less arguments. This is also called **currying**.

Chapter 11

Data Structures

Abstract A data structure is an organization of collections of data, such that operations on them are efficient. In earlier chapters, we have already looked at some examples, e.g.,

- strings which are variable length sequences of characters and which were discussed in Chapter 3,
- tuples which are fixed length sequences of values of variable types and which were discussed in Section 5.1,
- lists which are variable length sequences of values of identical type and which were discussed in Chapter 7, and
- stacks which are specialized lists, where values can only be added and removed from its head, and which was discussed in Section 9.4.

In this chapter, we will further consider the following data structures:

- *queues* which are specialized lists, where elements can be added to the end of the list and removed from its head,
- *trees* which are hierarchical orderings of data of a variable number of values of identical types,
- *sets* which are an unordered collection of unique values of the identical type, and
- *hash maps* which are mappings between sets of keys into sets of values.

These data structures have a long history and are often discussed from an abstract point of view in terms of their conceptual interface and from a computational complexity point of view, where details of their implementation are stressed. The above-mentioned data structures occur frequently alone or in combination with many programming solutions and form a solid basis for solving problems by programming. Some of these data structures have their predefined modules in F# but not all. In this chapter, we will give a brief introduction to each.

11.1 Queues

A queue is a sequence values that can be added to its end and removed from its front as illustrated in Figure 11.1. Queues appear often in real life: Standing in line at a

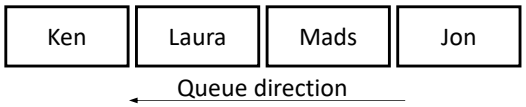


Fig. 11.1 A queue with Ken in the front and Jon in the back of the queue.

shop counter, orders await in a queue for their turn to be shipped in an online shop, and students waiting to be examined at an oral examination. Many operations on queues can be defined, but the following are always present in some form:

create: Create an empty queue.

enqueue: Add an element to the back of the queue.

dequeue: Remove the element at the front of the queue.

head: Get the value of the front element of the queue.

isEmpty: Check if the queue is empty.

As of the writing of this book, the standard collection of Fsharp libraries does not include a queues module, but they can easily be implemented using lists. For example, a queue of integers is implemented in Listing 11.1. This is a functional queue because the enqueue and dequeue operations return a new queue they are called, without destroying the old queue. Mutable queues are more common, where en- and dequeuing update the value of a queue as a side-effect. See Section 13.1 for more on mutable values. A simple application using this queue is shown in Listing 11.2 Note that this implementation, the computational complexity of all but `enqueue` is $O(1)$, while `enqueue` is $O(n)$, where n is the length of the list, since it relies on list concatenation. Faster implementations exist but are beyond the scope of this book.

11.2 Trees

A tree is a hierarchical organization of data. For example the expression

Listing 11.1 queue.fs:
Implementing a functional queue using lists.

```
module Queue

type element = int
type queue = element list

/// the empty queue
let create () : queue = []
/// add an element at the end of a queue
let enqueue (e: element) (q: queue) : queue =
  q @ [e]
/// remove the element at the front of the queue
let dequeue (q: queue) : (element option) * queue =
  match q with
    [] -> (None, [])
    | e::rst -> (Some e, rst)
/// the value at the front of the queue
let head (q: queue) : element option =
  q |> dequeue |> fst
/// check if the queue is empty
let isEmpty (q:queue): bool =
  q.IsEmpty
```

Listing 11.2 queueApp.fsx:
An application of the Queue module.

```
open Queue

let q = create () |> enqueue 3 |> enqueue 1
printfn "q = %A" q
printfn "Is q empty? %A" (isEmpty q)
let (v,newQ) = dequeue q
printfn "dequeue q = (%A, %A)" v newQ
```

```
$ dotnet fsi queue.fs queueApp.fsx
q = [3; 1]
Is q empty? false
dequeue q = (Some 3, [1])
```

$$\frac{1}{3+5} \tag{11.1}$$

can be represented as shown in Figure 11.2 Further examples of trees are the file structure on your hard disk, where a directory contains files and other directories, and the list of contents of this book, which has chapters consisting of sections which in turn consist of subsections.

Trees consists of *nodes* and *relations*. In Figure 11.2, 1, 3, 5, “Div”, and “Plus” are nodes and their relation are shown with arrows. Relations are often described as

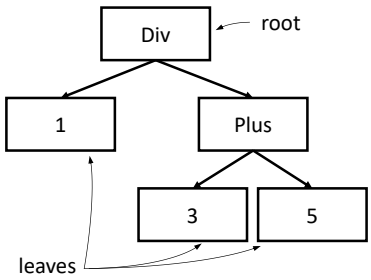


Fig. 11.2 A tree representation of $1/(3+5)$.

family relations, such that, e.g., 1 and "Plus" are *siblings* and are the *children* of the *parent* "Div". Nodes which do not have *descendants* are called *leaves*, and there must be on node, which does not have *ancestors* and that is called the *root*. Trees are often classified as being *k-ary*, if each node has at most k children. The example in Figure 11.2 is an example of a 2-ary or *binary tree*. Trees are often displayed with the children below their parent, in which case the arrowheads are neglected.

As of the writing of this book, the standard collection of Fsharp libraries does not include a tree module. Trees are somewhat complicated to program in the functional paradigm since they are non-linear structures. One way to represent them is with the use of discriminated union, as demonstrated in Listing 11.3. Such values are

Listing 11.3 bTree.fsx:
Representing a computational expression as a binary tree

```
type element = Value of float | Mul | Plus | Minus | Div
// A binary tree
type bTree = Leaf of element | Node of element * bTree * bTree
/// The tree representation of 1/(3+5)
let expr: bTree = Node (Div, Leaf (Value 1.0), Node (Plus,
   Leaf (Value 3.0), Leaf (Value 5.0)))
```

not easy to read, since it relies on nested types and tuples. Luckily, it is not difficult to make a function, which converts the tree into a string for later displaying on the screen. Such functions are commonly called `toString` and will be discussed later in Chapter 15. Here, the strategy will be to increase indentation proportional to the depth of the nodes printed, and one version of `toString` is shown in Listing 11.4 The result is to be interpreted as `Div` is the division operation of a `Value 1.0)` `and the result of performing the \lstinline{Plus` of two other values.

The `toString` function is an example of tree traversal. For binary trees 3 different traversal orders are common: *Prefix*, *infix* and *postfix* order according to the order in which the value of a node and its two children are handled. Consider a node with a value `elm` and its two children `left` and `right`, the ordering is as follows:

Prefix order: `elm`, `left`, and `right`

Listing 11.4 bTreeToString.fsx:
A `toString` method aids the interpretation of tree values.

```
type element = Value of float | Mul | Plus | Minus | Div
// A binary tree
type bTree = Leaf of element | Node of element * bTree * bTree
let rec toString (tab: string) (t:bTree) : string =
  match t with
    Leaf v -> tab+(string v)+"\n"
    | Node (op, a, b) ->
      tab + (string op) + "\n"
      + (toString (tab + "  ") a)
      + (toString (tab + "  ") b)

/// The tree representation of 1/(3+5)
let expr: bTree = Node (Div, Leaf (Value 1.0), Node (Plus,
   Leaf (Value 3.0), Leaf (Value 5.0)))
printf "%s" (toString "" expr)
```

```
$ dotnet fsi bTreeToString.fsx
Div
  Value 1.0
  Plus
    Value 3.0
    Value 5.0
```

Infix order: `left`, `elm`, and `right`

Postfix order: `left`, `right`, and `elm`

Thus, `toString` in Listing 11.4 is a prefix traversal. In Listing 11.5 the 3 traversal schemes are used to convert a binary tree to a list of `element` values.

Traversal methods linearize the tree structure and in general, 2 different traversals are required to reconstruct the original tree. However, as we saw in Section 9.4, a stack can be used to properly evaluate postfix representations of mathematical expressions. The code in Listing 9.17 can be modified for this purpose. Firstly, no computations need to be done, but every operator must result in a node and every value in a leaf, and instead of a stack of values, we must stack the sub-trees. The result is shown in Listing 11.6. Note that the resulting element on the stack is the original tree as expected.

Listing 11.5 bTreeTraversal.fsx:
Pre-, in-, and postfix traversal of a binary tree.

```
type element = Value of float | Mul | Plus | Minus | Div
type bTree = Leaf of element | Node of element * bTree * bTree

/// Prefix traversal of a binary tree
let rec prefix (t: bTree) : (element list) =
  match t with
    Leaf e -> [e]
    | Node (e, left, right) ->
      e :: (prefix left) @ (prefix right)
/// Infix traversal of a binary tree
let rec infix (t: bTree) : (element list) =
  match t with
    Leaf e -> [e]
    | Node (e, left, right) ->
      (infix left) @ [e] @ (infix right)
/// Postfix traversal of a binary tree
let rec postfix (t: bTree) : (element list) =
  match t with
    Leaf e -> [e]
    | Node (e, left, right) ->
      (postfix left) @ (postfix right) @ [e]

let expr: bTree = Node (Div, Leaf (Value 1.0), Node (Plus,
   Leaf (Value 3.0), Leaf (Value 5.0)))
printfn "Prefix traversal:\n%A" (prefix expr)
printfn "Infix traversal:\n%A" (infix expr)
printfn "Postfix traversal:\n%A" (postfix expr)
```

```
$ dotnet fsi bTreeTraversal.fsx
Prefix traversal:
[Div; Value 1.0; Plus; Value 3.0; Value 5.0]
Infix traversal:
[Value 1.0; Div; Value 3.0; Plus; Value 5.0]
Postfix traversal:
[Value 1.0; Value 3.0; Value 5.0; Plus; Div]
```

11.3 Programming intermezzo: Sorting Integers with a Binary Tree

Consider the problem of sorting a list of integers

Problem 11.1

Given a list of integers, such as `[5; 5; 7; 8; 3; 1; 8; 9; 0; 8]`, sort them and print the result on the screen.

Listing 11.6 bTreeFromPostfix.fsx:
Using a stack to reconstruct a tree from its postfix traversal.

```
open Stack

type element = Value of float | Mul | Plus | Minus | Div
type bTree = Leaf of element | Node of element * bTree * bTree

/// Postfix traversal of a binary tree
let rec postfix (t: bTree) : (element list) =
  match t with
    Leaf e -> [e]
    | Node (e, left, right) ->
      (postfix left) @ (postfix right) @ [e]
/// Convert postfix traversal back into a tree
let rec fromPostfix (tkns: element list) (stck:
   stack<bTree>): stack<bTree> =
  match tkns with
    [] -> stck
    | elm::rst ->
      match elm with
        Value v ->
          push (Leaf (Value v)) stck |> fromPostfix rst
        | _ ->
          let (a, stck1) = pop stck
          let (b, stck2) = pop stck1
          push (Node (elm, b, a)) stck2 |> fromPostfix rst

let src: bTree = Node (Div, Leaf (Value 1.0), Node (Plus,
   Leaf (Value 3.0), Leaf (Value 5.0)))
let psfx = postfix src
let tg = fromPostfix psfx (create ())
printfn "Original tree:\n%A" src
printfn "Postfix traversal:\n%A" psfx
printfn "From Postfix tranversal:\n%A" tg
```

```
$ dotnet fsi postfixLibraryGeneric.fs bTreeFromPostfix.fsx
Original tree:
Node (Div, Leaf (Value 1.0), Node (Plus, Leaf (Value 3.0),
   Leaf (Value 5.0)))
Postfix traversal:
[Value 1.0; Value 3.0; Value 5.0; Plus; Div]
From Postfix tranversal:
[Node (Div, Leaf (Value 1.0), Node (Plus, Leaf (Value 3.0),
   Leaf (Value 5.0)))]
```

This is such a common task that the list module contains a function for just this, `List.sort`, but here we will use the problem to study programming with binary trees. Our strategy will be to build a generic library for binary trees and an application with a function for sorting these values.

A strategy for sorting values is to enter them into a binary tree such that a node's value is always larger than its left child and less than or equal to its right child. The list could thus result in the tree shown in Figure 11.3. A consequence of the sorting

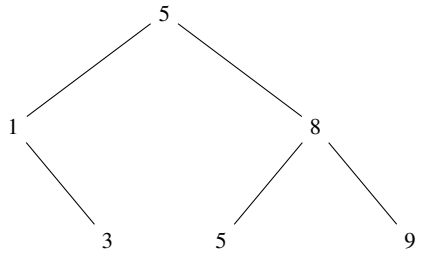


Fig. 11.3 A binary sorting tree for the list `[5; 1; 8; 3; 9; 5]`.

rule is that given a node, then its value is larger than all the values in the left sub-tree and no bigger than any values in the right sub-tree. Note also that the infix traversal of the tree gives a sorted list of values. In Figure 11.3 this is `[1; 3; 5; 5; 8; 9]`. In the spirit of our 8-step guide, let us make a sketch of the sorting algorithm to get a feeling of which types and functions could be useful. Firstly, we will need a generator of random numbers, to make sure we test many different combinations. This can be achieved with `System.Random()` and `List.map` as shown in Listing 11.7 Then we

Listing 11.7 bTreeSort.fsx:
Generating a list of random integers in the interval 0 to 9.

```
16 let rnd = System.Random()
17 let unsrt = List.map (fun _ -> rnd.Next 10) [1..10]
```

will need a type for a tree, and here we will use a generic type `bTree<'a>`, such that we reuse it for other values that can be compared with the $\leq$ operator. Our idea is to take the elements in the unsorted list to be inserted into a tree sequentially, and then finally write the result using infix traversal, e.g., as shown in Listing 11.8 Working

Listing 11.8 bTreeSort.fsx:
Using `List.fold` to sequentially insert integers into a binary sorting tree.

```
18 let srt = List.fold sortInsert (create unsrt.Head) unsrt.Tail
19 printfn "Unsorted list:\n%A\nSorted:\n%A" unsrt (infix srt)
```

with this, we realize that we will need a method for creating tree nodes and inserting them in a sorted fashion into a tree. Hence, we suggest the functions

```
create: v: 'a -> bTree<'a>
sortInsert: acc: bTree<'a> -> elm: 'a -> bTree<'a>
```

Given a value, the function `sortInsert` must start at the root of the tree and traverse down the tree until it finds a node with space for a leaf that obeys the sorting rule.

Thinking about this, we realize, that the tree type could well make use of the option type, e.g.,

```
type bTree<'a> = Node of 'a * bTree<'a> option * bTree<'a> option
```

and thus, an available position can be noted by the branch having the `None` value. Further, since we are in the functional programming paradigm, traversal must be recursive and insertion means the creation of a new tree. Finally, we arrived at the code in Listing 11.9. When programming this, it seemed useful to have

Listing 11.9 bTreeSort.fsx:
Insert a new node into an existing tree in a sorted manner.

```
3  let rec sortInsert (acc: bTree<int>) (elm: int): bTree<int> =
4    let v = retrieveValue acc
5    let l = tryRetrieveLeft acc
6    let r = tryRetrieveRight acc
7    if elm < v then
8      match l with
9        None -> replaceLeft (create elm) acc
10       | Some t -> replaceLeft (sortInsert t elm) acc
11   else
12     match r with
13       None -> replaceRight (create elm) acc
14       | Some t -> replaceRight (sortInsert t elm) acc
```

```
replaceLeft: c: bTree<'a> -> t: bTree<'a> -> bTree<'a>
replaceRight: c: bTree<'a> -> t: bTree<'a> -> bTree<'a>
retrieveValue: t: bTree<'a> -> 'a
tryRetrieveLeft: t: bTree<'a> -> bTree<'a> option
tryRetrieveRight: t: bTree<'a> -> bTree<'a> option
```

where `replaceLeft` and `replaceRight` replaces the left and right child `c` respectively in node `t`, `retrieveValue` retrieves the value stored in node `t`, and `tryRetrieveLeft` `tryRetrieveRight` follows the tradition of the other F# modules and returns a `Some bTree<'a>` or `None` depending on the existence of the child node.

At this point, we make a signature file for the library, and by the above arguments, we arrived at the code in Listing 11.10. This signature file specifically targets the problem of binary tree sorting, and a general-purpose library would have other functions as well. However, here we restrict the development to the problem at hand. The implementation of these functions turns out to be simple. In Listing 11.11, we have used pattern recognition in the definition of the arguments in several of the functions as, e.g., in `retrieveValue`. This makes the implementation particularly short, however, it may be less readable. Finally, putting it all together, a demonstration

Listing 11.10 bTreeGeneric.fsi:
The signature for a binary tree with insertion sort.

```
module bTree

type bTree<'a>

/// Create a tree with one value and no children
val create: 'a -> bTree<'a>
/// replace t's left child with c
val replaceLeft: c: bTree<'a> -> t: bTree<'a> -> bTree<'a>
/// replace t's right child with c
val replaceRight: c: bTree<'a> -> t: bTree<'a> -> bTree<'a>
/// retrieve the value of the root of m
val retrieveValue: t: bTree<'a> -> 'a
/// retrieve the left child
val tryRetrieveLeft: t: bTree<'a> -> bTree<'a> option
/// retrieve the right child
val tryRetrieveRight: t: bTree<'a> -> bTree<'a> option
/// Traverse the tree in infix order.
val infix: bTree<'a> -> 'a list
```

of the library and application can be seen in Listing 11.12. Due to the functional programming style, this implementation is robust and versatile, but not optimal. Depending on the internal workings of F#, each insertion potentially copies the pre-

Listing 11.11 bTreeGeneric.fs:
Insert a new node into an existing tree in a sorted manner.

```
module bTree

type bTree<'a> = Node of 'a * bTree<'a> option * bTree<'a>
   option

let create (v: 'a) =  Node (v, None, None)

let replaceLeft (c: bTree<'a>) (Node (v, l, r)) : bTree<'a> =
  Node (v, Some c, r)

let replaceRight (c: bTree<'a>) (Node (v, l, r)) : bTree<'a> =
  Node (v, l, Some c)

let retrieveValue (Node (v, l, r)) : 'a = v

let tryRetrieveLeft (Node (v, l, r)) : bTree<'a> option = l

let tryRetrieveRight (Node (v, l, r)) : bTree<'a> option = r

let rec infix (Node (v, l, r)) : ('a list) =
  (l |> Option.map infix |> Option.defaultValue [])
  @ [v]
  @ (r |> Option.map infix |> Option.defaultValue [])
```

Listing 11.12: The result of applying insertSorted on a random list.

```
$ dotnet fsi bTreeGeneric.fsi bTreeGeneric.fs bTreeSort.fsx
Unsorted list:
[9; 2; 1; 6; 6; 0; 7; 0; 6; 5]
Sorted:
[0; 0; 1; 2; 5; 6; 6; 6; 7; 9]
```

insertion tree and its sub-trees many times, and it is thus not to be expected to be particularly fast or memory-conserving.

11.4 Sets

A *set* is an unordered collection of data. Sets form the bases for much mathematics and much of computer science. For instance, `int` is the set of integers from $[2^16 \ldots 2^16.1]$ and `bool` is the set {false, true}. In fact, all types can be considered sets. Sets can be *empty*, a set with just 1 element is called a *singleton set*, and at least conceptually, sets can contain an infinite number of elements. In F# it is possible to represent *infinite sets*, but a discussion of this is beyond the scope of this book. The mathematical notation for sets is well-developed:

Empty set: The empty set is denoted $\emptyset$.

Set roster: A set can be written as a list of values with curly brackets and ellipses, e.g., $\{1, 2, \ldots, 10\}$. Remember, though, that the order of the elements is meaningless for sets.

Membership: An element x is a member in a set X is written as $x \in X$ and equivalently $x \notin X$ denotes non-membership.

Subsets: Subsets are denoted by $\subset$ and $\subseteq$, e.g., $\{'c', 'a',\} \subset \{'a', 'b', \ldots, 'z'\}$ and $\{'c', 'a',\} \subseteq \{'c', 'a',\}$. The negations of these are similarly defined $\{'a', 'b', \ldots, 'z',\} \not\subset \{'a', 'c'\}$.

Cardinality: The cardinality $|X|$ also known as the size of the set is the number of elements in the set X.

With sets comes a small number of basic operators

Complement: The complement of a subset $x \subset X$ is what is missing, i.e., $x^c = \{y : y \in X \text{ and } y \notin x\}$

Union: The union of two sets X and Y is $X \cup Y = \{z : z \in X \text{ or } z \in Y\}$

Intersection: The intersection of two sets X and Y is $X \cap Y = \{z : z \in X \text{ and } z \in Y\}$

Difference: The set difference between two sets X and Y is $X \setminus Y = \{z : z \in X \text{ and } z \notin Y\}$

F# has both a set class. Classes and objects will be discussed in further detail in Chapter 15. Presently, it is sufficient to think of a set class as an immutable type. F# furhter has a set module with more functions for sets, see `https://fsharp.github.io/fsharp-core-docs/reference/fsharp-collections-setmodule.html` for more details.

With the set module, an empty set can be created with `Set.empty`, and set can be created from a list `lst` as `Set lst`. Some of the important functions in the set module are:

`Set.add`: `x: 'T -> X: Set<'T> -> Set<'T>`
returns a new set $\{x\} \cup X$.

`Set.contains`: `x: 'T -> X: Set<'T> -> bool`
checks $x \in X$.

`Set.count`: `X: Set<'T> -> int`
returns $|X|$.

`Set.difference`: `X: Set<'T> -> Y: Set<'T> -> Set<'T>`
returns a new set $X \setminus Y$.

`Set.intersect`: `X: Set<'T> -> Y: Set<'T> -> Set<'T>`
returns a new set $X \cap Y$.

`Set.isEmpty`: `X: Set<'T> -> bool`
checks whether $X = \emptyset$.

`Set.isSubset`: `X: Set<'T> -> Y: Set<'T> -> bool`
checks whether $X \subset Y$.

`Set.remove`: `x: 'T -> X: Set<'T> -> Set<'T>`
returns a new set where $x \notin X$.

`Set.union`: `X: Set<'a> -> Y: Set<'a> -> Set<'a>`
returns a new set $X \cup Y$.

Sets can only be defined for elements, which can be compared, i.e., for which the $\geq$ and $\leq$ family of operators are defined.

Sets can be created from other collection types, such as lists, or by adding elements individually to an empty set, as demonstrated in Listing 11.13 A quick demonstration

Listing 11.13 setCreate.fsx:
Creating sets from lists or by adding elements one at a time.

```
let s1 = Set [3..5]
let s2 = Set.empty |> Set.add 3 |> Set.add 4 |> Set.add 5
printfn "s1 = %A\ns2 = %A" s1 s2
```

```
$ dotnet fsi setCreate.fsx
s1 = set [3; 4; 5]
s2 = set [3; 4; 5]
```

of union, intersection, and set difference is given in Listing 11.14. The module also

Listing 11.14 setUnionIntersectionDifference.fsx:
Illustration of set union, intersection, and difference.

```
let s1 = Set [3..5]
let s2 = Set [4;6;1]
printfn "s1 = %A\ns2 = %A" s1 s2
printfn "s1 union s2 = %A" (Set.union s1 s2)
printfn "s1 intersect s2 = %A" (Set.intersect s1 s2)
printfn "s1 difference s2 = %A" (Set.difference s1 s2)
```

```
$ dotnet fsi setUnionIntersectionDifference.fsx
s1 = set [3; 4; 5]
s2 = set [1; 4; 6]
s1 union s2 = set [1; 3; 4; 5; 6]
s1 intersect s2 = set [4]
s1 difference s2 = set [3; 5]
```

contains `Set.fold`, `Set.foldBack`, `Set.map`, and other functions similar to the `List` module, however, we leave it to the reader to consult the official documentation of the module for further detail.

11.5 Maps

A *map* is a discrete collection of relations between a domain and a codomain. As such, maps are discrete functions, but often they are termed databases or libraries since the elements from the domain are often called keys, and the corresponding values in the codomain are called values. The set of keys must be unique, while this is not the case for the set of values. Thus, the mapping between the set of keys and the set of values in a given map is *surjective* but not necessar-

ily *injective*. in F#, maps are sets of immutable *key-value pairs*. Similarly to sets, maps are classes and supported by a map module. A map can be created by `Map [("copenhagen", 1153615); ("berlin",3426354)]` or by `Map.empty |> Map.add "copenhagen" 1153615 |> Map.add "berlin" 3426354`. A brief list of important functions from the map module is:

`Map.add`: `k: 'K -> v: 'V -> m: Map<'K,'V> -> Map<'K,'V>`
: return a new map, which includes the `(k,v)` pair to the map `m`. If the key exists, then the existing key-value pair is replaced.

`Map.count`: `m: Map<'K,'V> -> int`
: count the number of `(k,v)` pairs there are in `m`.

`Map.isEmpty`: `m: Map<'K,'V> -> bool`
: checks whether the map `m` is empty.

`Map.keys`: `m: Map<'K,'V> -> System.Collections.Generic.ICollection<'K>`
: returns the sequence of keys in the map `m`. This can be turned into a set by `Map.keys m |> Set`.

`Map.remove`: `k: 'K -> m: Map<'K,'V> -> Map<'K,'V>`
: return a new map, which does not contain a `(k,v)` pair.

`Map.tryFind`: `k: 'K -> m: Map<'K,'V> -> 'V option`
: return the value `v` of the `(k,v)` pair if it exists.

`Map.values`: `Map<'K,'V> -> System.Collections.Generic.ICollection<'V>`
: returns the sequence of keys in the map `m`. This can be turned into a list by `Map.keys m |> List.ofSeq` or a set of unique values by `Map.keys m |> Set`.

Like sets, maps can only be defined for keys, which can be compared, i.e., for which the $\geq$ and $\leq$ family of operators are defined.

As an example of a map, consider the problem of producing the histogram of characters in a text. In Listing 11.15. The module also contains `Map.fold`, `Map.foldBack`, `Map.map`, and other functions similar to the `List` and the `Set` modules, however, we leave it to the reader to consult the official documentation of the module for further detail.

Listing 11.15 mapHistogram.fsx:
Calculating the histogram of characters using a map.

```
let rec hist (lst: char list) (m: Map<char,int>) :
    Map<char,int> =
  match lst with
    [] -> m
    | c::rst ->
      match (Map.tryFind c m) with
        None -> hist rst (Map.add c 1 m)
        | Some v -> hist rst (Map.add c (v+1) m)

let txt = "many years ago, there was an emperor, who was so
    excessively fond of new clothes, that he spent all his
    money in dress."
let lst = Seq.toList txt // Converts to a list of characters
let h = hist lst Map.empty

printfn "The text %A has the histogram\n%A" txt h
```

```
$ dotnet fsi mapHistogram.fsx
The text "many years ago, there was an emperor, who was so
    excessively fond of new clothes, that he spent all his
    money in dress." has the histogram
map
  [(' ', 22); (',', 3); ('.', 1); ('a', 8); ('c', 2); ('d',
    2); ('e', 14);
    ('f', 2); ('g', 1); ...]
```

11.6 Key concepts and terms in this chapter

In this chapter, we have looked at more data structures commonly used in programs. Key concepts have been:

- **Data structures** are often used as **models** of the real world and are defined **abstractly**. They may have many different **implementation** which vary in **computational complexity**.
- **Queues** are lists, where elements are added to the back and removed from the front. It does not have a built-in module in F# but is easy to implement.
- A **tree** is a non-linear structure, which organize data in a **hierarchical** structure. Trees are also not found as a standard data structure in F#, and unfortunately, not all facets of trees are easily implemented in the functional paradigm.
- A tree is a recursive data structure, consisting of **nodes** and their **children**, which are also trees.

- A **binary tree** is a node with at most two children.
- Binary trees are typically **traversed** in **prefix**, **infix**, or postfix order.
- A **set** is an immutable collection of values and is well-supported F#. Key set operators are **intersection**, **union**, and **set difference**.
- A **map** is a discrete, **surjective** but not necessarily **injective** function between a set of **keys** and **values**.

Part III

Imperative Programming Paradigms

In this part, we will primarily consider the *imperative* and *object-oriented programming paradigms*. Unfortunately, the imperative paradigm is used in the literature both to mean the overarching term of imperative paradigms and, as we do here, imperative programming without using object-oriented programming or other features. Thus it is ok, to say that object-oriented programming follows the imperative paradigm, but imperative programming does not necessarily follow the object-oriented paradigm.

Imperative programming is a paradigm for programming *states*. In imperative programming, the focus is on how a problem is to be solved, as a list of *statements* that affect states. In F#, states are *mutable values*, and they are affected by functions. In imperative programming, functions are sometimes called *procedures*, to emphasize that they may have *side-effects*. A side-effect is the result of the change of a state on the computer not related to the list of return parameters from the procedure. An imperative program is typically identified as using:

Mutable values
: Mutable values are holders of states, they may change over time, and thus have dynamic scope.

Procedures
: Procedures are functions that return "`()`", as opposed to functions that transform data. They are the embodiment of side-effects.

Side-effects
: Side-effects are changes of state that are not reflected in the arguments and return values of a function. The `printf` is an example of a procedure that writes text on the terminal but returns "`()`".

Loops
: The `for`- and `while`-loops typically use an iteration value to update some state, e.g., `for`-loops are often used to iterate through a list and summarize its contents.

Functional programming, can be seen as a subset of imperative programming and is discussed in Part II. *Object-oriented programming* is an extension of imperative programming, where statements and states are grouped into classes. For a discussion on object-oriented programming, see Chapters 15 to 17. Almost all computer hardware is designed for *machine code*, which is a common term used for many low-level computer programming languages, and almost all machine languages follow the imperative programming paradigm.

A prototypical example of an imperative program is a baking recipe, e.g., to make a loaf of bread, do the following:

1. Mix yeast with water.

2. Stir in salt, oil, and flour.
3. Knead until the dough has a smooth surface.
4. Let the dough rise until it has doubled its size.
5. Shape dough into a loaf.
6. Let the loaf rise until it is approximately double in size.
7. Bake in the oven until the bread is golden brown.

Each line in this example consists of one or more statements that are to be executed, and while executing them, states such as the size of the dough and the color of the bread change. Some execution will halt execution until certain conditions of these states are fulfilled, e.g., the bread will not be put into the oven for baking before it has risen sufficiently.

Object-oriented programming is a paradigm for encapsulating data and methods into cohesive units. E.g., a car can be modeled as an object, where data about the car including the amount of fuel, position, velocity, passengers, etc., can be stored together with functions for manipulating this data, e.g., move the car, add or extract passengers, etc. Key features of object-oriented programming are:

Encapsulation
: Data and methods are collected into a cohesive unit, and an application program need only focus on how to use the object, not on its implementation details.

Inheritance
: Objects are organized in a hierarchy of gradually increased specialties. This promotes a design of code that is of general use and code reuse.

Polymorphism
: By overriding methods from a base class, derived classes define new data types while their methods still produce results compatible with the base class definitions.

Object-oriented programming has a well-developed methodology for analysis and design. The analysis serves as input to the design phase, where the analysis reveals *what* a program is supposed to do, and the design *how* it is supposed to be doing it. The analysis should be expressed in general terms irrespective of the technological constraints, while the design should include technological constraints such as defined by the targeted language and hardware.

Chapter 12

Working With Files

Abstract Most programs do not live in isolation but interacts with the world around it. For example, many programs read and write from and to files or the internet. This is in general known as input and output. We have previously seen how to interact with the user by reading and writing to the terminal. In this chapter you will learn how to:

- Giving programs to ability to take arguments from the terminal,
- Read and write from to and from files,
- Handle errors by catching exceptions and if needed, throwing them yourself, and

Particularly with the topic of exceptions, we are leaving the domain of functional programming, and where errors before to a large extend were caught before a given program was run, we will now learn how to write programs, that can handle errors while they are running.

An important part of programming is handling data. A typical source of data is hard-coded bindings and expressions from libraries or the program itself, and the result is often shown on a screen as text output on the console. This is a good starting point when learning to program, and one which we have relied heavily upon in this book until now. However, many programs require more: We often need to ask a user to input data via, e.g., typing text on a keyboard, clicking with a mouse, or striking a pose in front of a camera. We also often need to load and save data to files, retrieve and deposit information from the internet, and visualize data graphically, as sounds, or by controlling electrical appliances. Here we will concentrate on working with the console, files, and the general concept of streams.

File and stream input and output are supported via built-in namespaces and classes. For example, `printf` is a family of functions defined in the `Printf` module of the `Fsharp.Core` namespace, and it is used to put characters on the `stdout` stream, i.e., to print on the screen. Likewise, `ReadLine` discussed in Section 2.4 is defined in the `System.Console` class, and it fetches characters from the `stdin` stream, that is, reads the characters the user types on the keyboard until newline is pressed.

A *file* on a computer is a resource used to store data in and retrieve data from. Files are often associated with a physical device, such as a hard disk, but can also be a virtual representation in memory. Files are durable, such that other programs can access them independently, given certain rules for access. A file has a name, a size, and a type, where the type is related to the basic unit of storage such as characters, bytes, and words, (`char`, `byte`, and `int32`). Often data requires a conversion between the internal format to and from the format stored in the file. E.g., floating point numbers are sometimes converted to a UTF8 string using `fprintf` in order to store them in a file in a human-readable form, and interpreted from UTF8 when retrieving them at a later point from the file. Files have a low-level structure, which varies from device to device, and the low-level details are less relevant for the use of the file and most often hidden for the user. Basic operations on files are *creation*, *opening*, *reading from*, *writing to*, *closing*, and *deleting*.

A *stream* is similar to files in that they are used to store data in and retrieve data from, but streams only allow for handling of data one element at a time, like the readout of a thermometer: we can make temperature readings as often as we like, making notes and thus saving a history of temperatures, but we cannot access the future. Hence, streams are in principle without an end, and thus have infinite size, and data from streams are programmed locally by considering the present and previous elements. In contrast, files are finite in size and allow for global operations on all the file's data. Files may be considered a stream, but the opposite is not true.

12.1 Command Line Arguments

Compiled programs may be started from the console with one or more arguments. E.g., if we have made a program called `prog`, then arguments may be passed as `mono prog arg1 arg2 ...`. To read the arguments in the program, we must define a function with the *`EntryPoint`* attribute, and this function must be of type `string array -> int`.

Listing 12.1: Defining an entry point function with arguments from the console.

```
[<EntryPoint>]
let <funcIdent> <arg> =
  <bodyExpr>
```

`<funcIdent>` is the function's name, `<arg>` is the name of an array of strings, and `<bodyExpr>` is the function body. Return value 0 implies a successful execution of the program, while a non-zero value means failure. The entry point function can only be in the rightmost file in the list of files given to `fsharpc`. An example is given in Listing 12.2. An example execution with arguments is shown in Listing 12.3.

Listing 12.2 commandLineArgs/Program.fs: Interacting with a user with `ReadLine` and `WriteLine`.

```
[<EntryPoint>]
let main args =
    printfn "Arguments passed to function : %A" args
    0  // Signals that program terminated successfully
```

In Bash, the return value is called the *exit status* and can be tested using Bash's `if`

Listing 12.3: An example dialogue of running Listing 12.2.

```
$ cd commandLineArgs; dotnet run Hello World
Arguments passed to function : [|"Hello"; "World"|]
```

statements, as demonstrated in Listing 12.4. Also in Bash, the exit status of the last

Listing 12.4: Testing return values in Bash when running Listing 12.2.

```
$ cd commandLineArgs; if dotnet run Hello World; then echo
   "success"; else echo "failure"; fi
Arguments passed to function : [|"Hello"; "World"|]
success
```

executed program can be accessed using the `$?` built-in environment variable. In Windows, this same variable is called `%errorlevel%`.

12.2 Interacting With the Console

From a programming perspective, the console is a stream: A program may send new data to the console, but cannot return to previously sent data and make changes. Likewise, the program may retrieve input from the user, but cannot go back and ask the user to have input something else, nor can we peek into the future and retrieve what the user will input in the future. The console uses three built-in streams in `System.Console`, listed in Table 12.1. On the console, the standard output

Stream	Description
`stdout`	Standard output stream used to display regular output. It typically streams data to the console.
`stderr`	Standard error stream used to display warnings and errors, typically streams to the same place as `stdout`.
`stdin`	Standard input stream used to read input, typically from the keyboard input.

Table 12.1 Three built-in streams in `System.Console`.

and error streams are displayed as text, and it is typically not possible to see a distinction between them. However, command-line interpreters such as Bash can, and it is possible from the command-line to filter output from programs according to these streams. However, a further discussion on this is outside the scope of this text. In `System.Console` there are many functions supporting interaction with the console, and the most important ones are shown in Table 12.2. Note that you must

Function	Description
`Write: string -> unit`	Write to the console. E.g., `System.Console.Write "Hello world"`. Similar to `printf`.
`WriteLine: string -> unit`	As `Write`, but followed by a newline character, e.g., `WriteLine "Hello world"`. Similar to `printfn`.
`Read: unit -> int`	Wait until the next key is pressed, and read its value. The key pressed is echoed to the screen.
`ReadKey: bool -> System.ConsoleKeyInfo`	As `Read`, but returns more information about the key pressed. When given the value `true` as argument, then the key pressed is not echoed to the screen. E.g., `ReadKey true`.
`ReadLine unit -> string`	Read the next sequence of characters until newline from the keyboard, e.g., `ReadLine ()`.

Table 12.2 Some functions for interacting with the user through the console in the `System.Console` class. Prefix "`System.Console.`" is omitted for brevity.

supply the empty argument "`()`" to the `Read` functions in order to run most of the functions instead of referring to them as values. A demonstration of the use of `Write`, `WriteLine`, and `ReadLine` is given in Listing 12.5. The functions `Write` and `WriteLine` act as `printfn` without a formatting string. These functions have

Listing 12.5 userDialogue.fsx:
Interacting with a user with `ReadLine` and `WriteLine`. The user typed "3.5" and "7.4".

```
System.Console.WriteLine "To multiply a and b"
System.Console.Write "Enter a: "
let a = float (System.Console.ReadLine ())
System.Console.Write "Enter b: "
let b = float (System.Console.ReadLine ())
System.Console.WriteLine ("a * b = " + string (a * b))
```

```
$ dotnet fsi userDialogue.fsx
To multiply a and b
Enter a: 3.5
Enter b: 7.4
a * b = 25.900000000000002
```

many overloaded definitions, the description of which is outside the scope of this book. **For writing to the console, `printf` is to be preferred.** ⋆

Often `ReadKey` is preferred over `Read`, since the former returns a value of type `System.ConsoleKeyInfo` which is a structure with three properties:

`Key`: A `System.ConsoleKey` enumeration of the key pressed. E.g., the character 'a' is `ConsoleKey.A`.

`KeyChar`: A unicode representation of the key.

`Modifiers`: A `System.ConsoleModifiers` enumeration of modifier keys shift, crtl, and alt.

An example of a dialogue is shown in Listing 12.6.

12.3 Exceptions

Exceptions are runtime errors, such as division by zero. E.g., attempting integer division by zero halts execution and a long somewhat cryptic error message is written to screen, as illustrated in Listing 12.7. The error message contains much information. The first part, `System.DivideByZeroException: Attempted to divide by zero` is the error-name with a brief ellaboration. Then follows a list libraries that were involved when the error occurred, and finally F# states that it `Stopped due to error`. `System.DivideByZeroException` is a built-in exception type, and the built-in integer division operator chooses to raise the exception when the undefined

Listing 12.6 readKey.fsx:
Reading keys and modifiers. The user pressed 'a', 'shift-a', and 'crtl-a', and the program was terminated by pressing 'crtl-c'. The 'alt-a' combination does not work on MacOS.

```
open System

printfn "Start typing"
while true do
  let key = Console.ReadKey true
  let shift =
    if key.Modifiers = ConsoleModifiers.Shift then "SHIFT+"
   else ""
  let alt =
    if key.Modifiers = ConsoleModifiers.Alt then "ALT+" else
   ""
  let ctrl =
    if key.Modifiers = ConsoleModifiers.Control then "CTRL+"
   else ""
  printfn "You pressed: %s%s%s%s" shift alt ctrl
   (key.Key.ToString ())
```

```
$ dotnet fsi readKey.fsx
Start typing
You pressed: A
You pressed: SHIFT+A
You pressed: CTRL+A
```

Listing 12.7: Division by zero halts execution with an error message.

```
> 3 / 0;;
System.DivideByZeroException: Attempted to divide by zero.
   at <StartupCode$FSI_0002>.$FSI_0002.main@() in
   /Users/jrh630/repositories/fsharp-book/src/stdin:line 1
```

division by zero is attempted. Many times such errors can be avoided by clever program design. However, this is not always possible or desirable, which is why F# implements exception handling for graceful control.

Exceptions are a basic-type called *exn*, and F# has a number of built-in ones, a few of which are listed in Table 12.3.

Attribute	Description
`ArgumentException`	Arguments provided are invalid.
`DivideByZeroException`	Division by zero.
`NotFiniteNumberException`	floating point value is plus or minus infinity, or Not-a-Number (NaN).
`OverflowException`	Arithmetic or casting caused an overflow.
`IndexOutOfRangeException`	Attempting to access an element of an array using an index which is less than zero or equal or greater than the length of the array.

Table 12.3 Some built-in exceptions. The prefix `System.` has been omitted for brevity.

Exceptions are handled by the `try`–keyword expressions. We say that an expression may *raise* or *cast* an exception and that the `try`–expression may *catch* and *handle* the exception by another expression.

Exceptions like in Listing 12.7 may be handled by `try`–`with` expressions, as demonstrated in Listing 12.8. In the example, when the division operator raises the

Listing 12.8 exceptionDivByZero.fsx:
A division by zero is caught and a default value is returned.

```
let div enum denom =
  try
    enum / denom
  with
    | :? System.DivideByZeroException -> System.Int32.MaxValue
printfn "3 / 1 = %d" (div 3 1)
printfn "3 / 0 = %d" (div 3 0)
```

```
$ dotnet fsi exceptionDivByZero.fsx
3 / 1 = 3
3 / 0 = 2147483647
```

`System.DivideByZeroException` exception, then `try`–`with` catches it and returns the value `System.Int32.MaxValue`. Division by zero is still an undefined operation, but with the exception system, the program is able to receive a message about this undefined situation and choose an appropriate action.

The `try` expressions comes in two flavors: `try`–`with` and `try`–`finally` expressions.

The *`try`–`with`* expression has the following syntax,

Listing 12.9: Syntax for the `try`–`with` exception handling.

```
try
   <testExpr>
with
   [ | ] <pat1> -> <exprHndl1>
   | <pa2> -> <exprHndl2>
   | <pat3> -> <exprHndl3>
   ...
```

where `<testExpr>` is an expression which might raise an exception, `<patn>` is a pattern, and `<exprHndln>` is the corresponding exception handler. The value of the `try`–expression is either the value of `<testExpr>`, if it does not raise an exception, or the value of the exception handler `<exprHndln>` of the first matching pattern `<patn>`. The above is using lightweight syntax. Regular syntax omits newlines.

In Listing 12.8 *dynamic type matching* is used using the "`:?`" lexeme, i.e., the pattern matches exception with type `System.DivideByZeroException` at runtime. The

exception value may contain furter information and can be accessed if named using the *as*–keyword, as demonstrated in Listing 12.10. Here the exception value is

Listing 12.10 exceptionDivByZeroNamed.fsx: Exception value is bound to a name. Compare to Listing 12.8.

```
let div enum denom =
  try
    enum / denom
  with
    | :? System.DivideByZeroException as ex ->
      printfn "Error: %s" ex.Message
      System.Int32.MaxValue

printfn "3 / 1 = %d" (div 3 1)
printfn "3 / 0 = %d" (div 3 0)
```

```
$ dotnet fsi exceptionDivByZeroNamed.fsx
3 / 1 = 3
Error: Attempted to divide by zero.
3 / 0 = 2147483647
```

bound to the name `ex`.

All exceptions may be caught as the dynamic type `System.Exception`, and F# implements a short-hand for catching an exception and binding its value to a name as demonstrated in Listing 12.11 Finally, the short-hand may be guarded with a

Listing 12.11 exceptionDivByZeroShortHand.fsx: An exception of type `System.Exception` is bound to a name. Compare to Listing 12.10.

```
let div enum denom =
  try
    enum / denom
  with
    | ex -> printfn "Error: %s" ex.Message;
   System.Int32.MaxValue

printfn "3 / 1 = %d" (div 3 1)
printfn "3 / 0 = %d" (div 3 0)
```

```
$ dotnet fsi exceptionDivByZeroShortHand.fsx
3 / 1 = 3
Error: Attempted to divide by zero.
3 / 0 = 2147483647
```

when–guard, as demonstrated in Listing 12.12. The first pattern only matches the `System.Exception` exception when `enum` is 0, in which case the exception handler returns 0.

Listing 12.12 exceptionDivByZeroGuard.fsx:
An exception of type `System.Exception` is bound to a name and guarded. Compare to Listing 12.11.

```
let div enum denom =
  try
    enum / denom
  with
    | ex when enum = 0 -> 0
    | ex -> System.Int32.MaxValue

printfn "3 / 1 = %d" (div 3 1)
printfn "3 / 0 = %d" (div 3 0)
printfn "0 / 0 = %d" (div 0 0)
```

```
$ dotnet fsi exceptionDivByZeroGuard.fsx
3 / 1 = 3
3 / 0 = 2147483647
0 / 0 = 0
```

Thus, if you don't care about the type of exception, then you need only use the short-hand pattern matching and name binding demonstrated in Listing 12.11 and Listing 12.12, but if you would like to distinguish between types of exceptions, then you must use explicit type matching and possibly value binding demonstrated in Listing 12.8 and Listing 12.10

The *try–finally* expression has the following syntax,

Listing 12.13: Syntax for the `try–finally` exception handling.

```
try
   <testExpr>
finally
   <cleanupExpr>
```

The `try–finally` expression evaluates the `<cleanupExpr>` expression following evaluation of the `<testExpr>`, regardless of whether an exception is raised or not, as illustrated in Listing 12.14. Here, the `finally` branch is evaluated following the evaluation of the test expression regardless of whether the test expression raises an exception or not. However, if an exception is raised in a `try–finally` expression and there is no outer `try–with` expression, then execution stops without having evaluated the `finally` branch.

Exceptions can be raised using the *raise*-function,

Listing 12.15: Syntax for the `raise` function that raises exceptions.

```
raise (<expr>)
```

Listing 12.14 exceptionDivByZeroFinally.fsx:
The `finally` branch is executed regardless of an exception.

```
let div enum denom =
  printf "Doing division:"
  try
    printf " %d %d." enum denom
    enum / denom
  finally
    printfn " Division finished."

printfn "3 / 1 = %d" (try div 3 1 with ex -> 0)
printfn "3 / 0 = %d" (try div 3 0 with ex -> 0)
```

```
$ dotnet fsi exceptionDivByZeroFinally.fsx
Doing division: 3 1. Division finished.
3 / 1 = 3
Doing division: 3 0. Division finished.
3 / 0 = 0
```

An example of raising the `System.ArgumentException` is shown in Listing 12.16 In this example, division by zero is never attempted and instead an exception is raised

Listing 12.16 raiseArgumentException.fsx:
Raising the division by zero with customized message.

```
let div enum denom =
  if denom = 0 then
    raise (System.ArgumentException "Error: \"division by
     0\"")
  else
    enum / denom

printfn "3 / 0 = %s" (try (div 3 0 |> string) with ex ->
    ex.Message)
```

```
$ dotnet fsi raiseArgumentException.fsx
3 / 0 = Error: "division by 0"
```

which must be handled by the caller. Note that the type of `div` is `int -> int -> int` because `denom` is compared with an integer in the conditional statement. This contradicts the typical requirements for `if` statements, where every branch has to return the same type. However, any code that explicitly raise exceptions are ignored, and the type is inferred by the remaining branches.

Programs may define new exceptions using the syntax,

Listing 12.17: Syntax for defining new exceptions.

```
exception <ident> of <typeId> {* <typeId>}
```

An example of defining a new exception and raising it is given in Listing 12.18. Here

Listing 12.18 exceptionDefinition.fsx:
A user-defined exception is raised but not caught by outer construct.

```
exception DontLikeFive of string

let picky a =
  if a = 5 then
    raise (DontLikeFive "5 sucks")
  else
    a

printfn "picky %A = %A" 3 (try picky 3 |> string with ex ->
   ex.Message)
printfn "picky %A = %A" 5 (try picky 5 |> string with ex ->
   ex.Message)
```

```
$ dotnet fsi exceptionDefinition.fsx
picky 3 = "3"
picky 5 = "Exception of type 'FSI_0001+DontLikeFive' was
   thrown."
```

an exception called `DontLikeFive` is defined, and it is raised in the function `picky`. The example demonstrates that catching the exception as a `System.Exception` as in Listing 12.11, the `Message` property includes information about the exception name but not its argument. To retrieve the argument `"5 sucks"`, we must match the exception with the correct exception name, as demonstrated in Listing 12.19.

F# includes the *`failwith`* function to simplify the most common use of exceptions. It is defined as `failwith : string -> exn` and takes a string and raises the built-in `System.Exception` exception. An example of its use is shown in Listing 12.20. To catch the `failwith` exception, there are several choices. The exception casts a `System.Exception` exception, which may be caught using the `:?` pattern, as shown in Listing 12.21. However, this gives annoying warnings, since F# internally is built such that all exception match the type of `System.Exception`. Instead, it is better to either match using the wildcard pattern as in Listing 12.22, or use the built-in `Failure` pattern as in Listing 12.23. Notice how only the `Failure` pattern allows for the parsing of the message given to `failwith` as an argument.

Invalid arguments are such a common reason for failures, that a built-in function for handling them has been supplied in F#. The *`invalidArg`* takes 2 strings and raises the built-in `ArgumentException`, as shown in Listing 12.24. The `invalidArg` function raises an `System.ArgumentException`, as shown in Listing 12.25.

Listing 12.19 exceptionDefinitionNCatch.fsx: Catching a user-defined exception.

```
exception DontLikeFive of string

let picky a =
  if a = 5 then
    raise (DontLikeFive "5 sucks")
  else
    a

try
  printfn "picky %A = %A" 3 (picky 3)
  printfn "picky %A = %A" 5 (picky 5)
with
  | DontLikeFive msg -> printfn "Exception caught with
   message: %s" msg
```

```
$ dotnet fsi exceptionDefinitionNCatch.fsx
picky 3 = 3
Exception caught with message: 5 sucks
```

Listing 12.20 exceptionFailwith.fsx: An exception raised by `failwith`.

```
if true then failwith "hej"
```

```
$ dotnet fsi exceptionFailwith.fsx
System.Exception: hej
   at <StartupCode$FSI_0001>.$FSI_0001.main@() in
   exceptionFailwith.fsx:line 1
Stopped due to error
```

The `try` construction is typically used to gracefully handle exceptions, but there are times where you may want to pass on the bucket, so to speak, and re-raise the exception. This can be done with the *`reraise`*, as shown in Listing 12.26. The `reraise` function is only allowed to be the final call in the expression of a `with` rule.

12.4 Storing and Retrieving Data From a File

A file stored on the filesystem has a name, and it must be opened before it can be accessed and closed when finished. Opening files informs the operating system that your program is now going to use the file. While a file is open, the operating system

Listing 12.21 exceptionSystemException.fsx:
Catching a `failwith` exception using type matching pattern.

```
let _ =
  try
    failwith "Arrrrg"
  with
    :? System.Exception -> printfn "So failed"
```

```
$ dotnet fsi exceptionSystemException.fsx


exceptionSystemException.fsx(5,5): warning FS0067: This type
   test or downcast will always hold



exceptionSystemException.fsx(5,5): warning FS0067: This type
   test or downcast will always hold

So failed
```

Listing 12.22 exceptionMatchWildcard.fsx:
Catching a `failwith` exception using the wildcard pattern.

```
let _ =
  try
    failwith "Arrrrg"
  with
    _ -> printfn "So failed"
```

```
$ dotnet fsi exceptionMatchWildcard.fsx
So failed
```

will protect it depending on how the file is opened. E.g., if you are going to write to

Listing 12.23 exceptionFailure.fsx:
Catching a `failwith` exception using the `Failure` pattern.

```
let _ =
  try
    failwith "Arrrrg"
  with
    Failure msg ->
      printfn "The castle of %A" msg
```

```
$ dotnet fsi exceptionFailure.fsx
The castle of "Arrrrg"
```

Listing 12.24 exceptionInvalidArg.fsx:
An exception raised by `invalidArg`. Compare with Listing 12.16.

```
if true then invalidArg "a" "is too much 'a'"
```

```
$ dotnet fsi exceptionInvalidArg.fsx
System.ArgumentException: is too much 'a' (Parameter 'a')
   at <StartupCode$FSI_0001>.$FSI_0001.main@() in
   exceptionInvalidArg.fsx:line 1
Stopped due to error
```

Listing 12.25 exceptionInvalidArgNCatch.fsx:
Catching the exception raised by `invalidArg`.

```
let _ =
  try
    invalidArg "a" "is too much 'a'"
  with
    :? System.ArgumentException -> printfn "Argument is no
   good!"
```

```
$ dotnet fsi exceptionInvalidArgNCatch.fsx
Argument is no good!
```

Listing 12.26 exceptionReraise.fsx:
Reraising an exception.

```
let _ =
  try
    failwith "Arrrrg"
  with
    Failure msg ->
      printfn "The castle of %A" msg
      reraise()
```

```
$ dotnet fsi exceptionReraise.fsx
The castle of "Arrrrg"
System.Exception: Arrrrg
   at <StartupCode$FSI_0001>.$FSI_0001.main@() in
   exceptionReraise.fsx:line 3
Stopped due to error
```

the file, then this typically implies that no one else may write to the file at the same time, since simultaneous writing to a file may leave the resulting file in an uncertain state. Sometimes the operating system will realize that a file that was opened by a program is no longer being used, e.g., since the program is no longer running,

★ but **it is good practice always to release reserved files, e.g., by closing them as soon as possible, such that other programs may have access to it.** On the other

hand, it is typically safe for several programs to read the same file at the same time, but it is still important to close files after their use, such that the operating system can effectively manage the computer's resources. Reserved files are just one of the possible obstacles that you may meet when attempting to open a file. Other points of failure may be that the file does not exist, your program may not have sufficient rights for accessing it, or the device where the file is stored may have unreliable access. Thus, **never assume that accessing files always works, but program defensively, e.g., by checking the return status of the file accessing functions and by `try` constructions.** ⋆

Data in files may have been stored in various ways, e.g., it may contain UTF8 encoded characters or sequences of floating point numbers stored as raw bits in chunks of 64 bits, or it may be a sequence of bytes that are later going to be interpreted as an image in jpeg or tiff format. To aid in retrieving the data, F# has a family of open functions, all residing in the `System.IO.File` class. These are described in Table 12.4.

`System.IO.File`	Description
`Open: (path : string) * (mode : FileMod -> FileStream`	Request the opening of a file on `path` for reading and writing with access mode `FileMode`, see Table 12.5. Other programs are not allowed to access the file before this program closes it.
`OpenRead: (path : string) -> FileStream`	Request the opening of a file on `path` for reading. Other programs may read the file regardless of this opening.
`OpenText: (path : string) -> StreamReader`	Request the opening of an existing UTF8 file on `path` for reading. Other programs may read the file regardless of this opening.
`OpenWrite: (path : string) -> FileStream`	Request the opening of a file on `path` for writing with `FileMode.OpenOrCreate`. Other programs may not access the file before this program closes it.
`Create: (path : string) -> FileStream`	Request the creation of a file on `path` for reading and writing, overwriting any existing file. Other programs may not access the file before this program closes it.
`CreateText: (path : string) -> StreamWriter`	Request the creation of an UTF8 file on `path` for reading and writing, overwriting any existing file. Other programs may not access the file before this program closes it.

Table 12.4 The family of `System.IO.File.Open` functions. See Table 12.5 for a description of `FileMode`, Tables 12.6 and 12.7 for a description of `FileStream`, Table 12.8 for a description of `StreamReader`, and Table 12.9 for a description of `StreamWriter`.

For the general `Open` function, you must also specify how the file is to be opened. This is done with a special set of values described in Table 12.5. An example of how a file is opened and later closed is shown in Listing 12.27. Notice how the example uses a defensive programming style, where the `try`-expression is used to return the optional datatype, and further processing is made dependent on the success of the opening operation.

FileMode	Description
Append	Open a file and seek to its end, if it exists, or create a new file. Can only be used together with FileAccess.Write. May throw IOException and NotSupportedException exceptions.
Create	Create a new file. If a file with the given filename exists, then that file is deleted. May throw the UnauthorizedAccessException exception.
CreateNew	Create a new file, but throw the IOException exception if the file already exists.
Open	Open an existing file. System.IO.FileNotFoundException exception is thrown if the file does not exist.
OpenOrCreate	Open a file, if it exists, or create a new file.
Truncate	Open an existing file and truncate its length to zero. Cannot be used together with FileAccess.Read.

Table 12.5 File mode values for the System.IO.Open function.

Listing 12.27 openFile.fsx:
Opening and closing a file, in this case, the source code of this same file.

```
let filename = "openFile.fsx"

let reader =
  try
    Some (System.IO.File.Open (filename,
     System.IO.FileMode.Open))
  with
    _ -> None

if reader.IsSome then
  printfn "The file %A was successfully opened." filename
  reader.Value.Close ()
```

```
$ dotnet fsi openFile.fsx
```

In F#, the distinction between files and streams is not very clear. F# offers built-in support for accessing files as bytes through the System.IO.FileStream class, and for characters in a particular encoding through the System.IO.TextReader and System.IO.TextWriter.

A successfully opened System.IO.FileStream file by, e.g., System.IO.File.OpenRead from Table 12.4, will result in an FileStream object. From this object we can extract information about the file, such as the permitted operations and more listed in Table 12.6. This information is important in order to restrict the operation that we will perform on the file. Some typical operations are listed in and 12.7. E.g., we may Seek a particular position in the file, but only within the range of legal postions from 0 until the length of the file. Most operating systems do not necessarily write information to files immediately after one of the Write functions, but will often for optimization purposes collect information in a buffer that is to be written to a device in batches. However, sometimes is is useful to be able to force the operating system to empty its buffer to the device. This is called *flushing* and can be forced using the Flush function.

Property	Description
`CanRead`	Gets a value indicating whether the current stream supports reading. (Overrides Stream.CanRead.)
`CanSeek`	Gets a value indicating whether the current stream supports seeking. (Overrides Stream.CanSeek.)
`CanWrite`	Gets a value indicating whether the current stream supports writing. (Overrides Stream.CanWrite.)
`Length`	Gets the length of a stream in bytes. (Overrides Stream.Length.)
`Name`	Gets the name of the FileStream that was passed to the constructor.
`Position`	Gets or sets the current position of this stream. (Overrides Stream.Position.)

Table 12.6 Some properties of the `System.IO.FileStream` class.

`Method`	Description
`Close ()`	Closes the stream.
`Flush ()`	Causes any buffered data to be written to the file.
`Read byte[] * int * int`	Reads a block of bytes from the stream and writes the data in a given buffer.
`ReadByte ()`	Read a byte from the file and advances the read position to the next byte.
`Seek int * SeekOrigin`	Sets the current position of this stream to the given value.
`Write byte[] * int * int`	Writes a block of bytes to the file stream.
`WriteByte byte`	Writes a byte to the current position in the file stream.

Table 12.7 Some methods of the `System.IO.FileStream` class.

Text is typically streamed through the `StreamReader` and `StreamWriter`. These may be considered higher-order stream processing, since they include an added

Listing 12.28 readFile.fsx:
An example of opening a text file and using the `StreamReader` properties and methods.

```
let rec printFile (reader : System.IO.StreamReader) =
  if not(reader.EndOfStream) then
    let line = reader.ReadLine ()
    printfn "%s" line
    printFile reader

let filename = "readFile.fsx"
let reader = System.IO.File.OpenText filename
printFile reader
```

```
$ dotnet fsi readFile.fsx
let rec printFile (reader : System.IO.StreamReader) =
  if not(reader.EndOfStream) then
    let line = reader.ReadLine ()
    printfn "%s" line
    printFile reader

let filename = "readFile.fsx"
let reader = System.IO.File.OpenText filename
printFile reader
```

interpretation of the bits to strings. A `StreamReader` has methods similar to a `FileStream` object and a few new properties and methods, such as the `EndOfStream` property and `ReadToEnd` method, see Table 12.8. Likewise, a `StreamWriter` has

Property/Method	Description
`EndOfStream`	Check whether the stream is at its end.
`Close ()`	Closes the stream.
`Flush ()`	Causes any buffered data to be written to the file.
`Peek ()`	Reads the next character, but does not advance the position.
`Read ()`	Reads the next character.
`Read char[] * int * int`	Reads a block of bytes from the stream and writes the data in a given buffer.
`ReadLine ()`	Reads the next line of characters until a newline. Newline is discarded.
`ReadToEnd ()`	Reads the remaining characters until end-of-file.

Table 12.8 Some methods of the `System.IO.StreamReader` class.

an added method for automatically flushing after every writing operation. A simple

Property/Method	Description
`AutoFlush : bool`	Gets or sets the auto-flush. If set, then every call to `Write` will flush the stream.
`Close ()`	Closes the stream.
`Flush ()`	Causes any buffered data to be written to the file.
`Write 'a`	Writes a basic type to the file.
`WriteLine string`	As `Write`, but followed by newline.

Table 12.9 Some methods of the `System.IO.StreamWriter` class.

example of opening a text-file and processing it is given in Listing 12.28. Here the program reads the source code of itself, and prints it to the console.

12.5 Working With Files and Directories.

F# has support for managing files, summarized in the `System.IO.File` class and summarized in Table 12.10.

Function	Description
`Copy (src : string, dest : string)`	Copy a file from `src` to `dest`, possibly overwriting any existing file.
`Delete string`	Delete a file
`Exists string`	Checks whether the file exists
`Move (from : string, to : string)`	Move a file from `src` to `to`, possibly overwriting any existing file.

Table 12.10 Some methods of the `System.IO.File` class.

In the `System.IO.Directory` class there are a number of other frequently used functions, summarized in Table 12.11.

Function	Description
`CreateDirectory string`	Create the directory and all implied sub-directories.
`Delete string`	Delete a directory.
`Exists string`	Check whether the directory exists.
`GetCurrentDirectory ()`	Get working directory of the program.
`GetDirectories (path : string)`	Get directories in `path`.
`GetFiles (path : string)`	Get files in `path`.
`Move (from : string, to : string)`	Move a directory and its content from `src` to `to`.
`SetCurrentDirectory : (path : string) -> unit`	Set the current working directory of the program to `path`.

Table 12.11 Some methods of the `System.IO.Directory` class.

In the `System.IO.Path` class there are a number of other frequently used functions summarized in Table 12.12.

Function	Description
`Combine string * string`	Combine two paths into a new path.
`GetDirectoryName (path: string)`	Extract the directory name from `path`.
`GetExtension (path: string)`	Extract the extension from `path`.
`GetFileName (path: string)`	Extract the name and extension from `path`.
`GetFileNameWithoutExtension (path : string)`	Extract the name without the extension from `path`.
`GetFullPath (path : string)`	Extract the absolute path from `path`.
`GetTempFileName ()`	Create a uniquely named and empty file on disk and return its full path.

Table 12.12 Some methods of the `System.IO.Path` class.

12.6 Programming intermezzo: Name of Existing File Dialogue

A typical problem when working with files is

Problem 12.1

Ask the user for the name of an existing file.

Such dialogues often require the program to aid the user, e.g., by telling the user which files are available, and by checking that the filename entered is an existing file.

We will limit our request to the present directory and use `System.Console.ReadLine` to get input from the user. Our strategy will be twofold. Firstly we will query the filesystem for the existing files using `System.IO.Directory.GetFiles`, and print these to the screen. Secondly, we will use `System.IO.File.Exists` to ensure that a file exists with the entered filename. We use the `Exists` function rather than examining the array obtained with `GetFiles`, since files may have been added or removed, since the `GetFiles` was called. A solution is shown in Listing 12.29. Note that it is programmed using a `while`-loop and with a flag `fileExists` used to exit the loop. The solution has a caveat: What should be done if the user decides not to enter a ★ filename at all. **Including a 'cancel'-option is a good style for any user interface, and should be offered when possible.** In a text-based dialogue, this would require us to use an input, which cannot be a filename, to ensure that all possible filenames and 'cancel'-option is available to the user. This problem has not been addressed in the code.

Listing 12.29 filenamedialogue.fsx:
Ask the user to input a name of an existing file.

```
let rec getAFileName () =
  System.Console.Write("Enter Filename: ")
  let filename = System.Console.ReadLine()
  if System.IO.File.Exists filename then filename
  else getAFileName ()

let listOfFiles = System.IO.Directory.GetFiles "."
printfn "Directory contains: %A" listOfFiles
let filename = getAFileName ()
printfn "You typed: %s" filename
```

12.7 Resource Management

Streams and files are examples of computer resources that may be shared by several applications. Most operating systems allow for several applications to be running in parallel, and to avoid unnecessarily blocking and hogging of resources, all responsible applications must release resources as soon as they are done using them. F# has language constructions for automatic releasing of resources: the `use` binding and the `using` function. These automatically dispose of resources when the resource's name binding falls out of scope. Technically, this is done by calling the *`Dispose`* method on objects that implement the *`System.IDisposable`* interface. See Section 16.4 for more on interfaces.

The *`use`* keyword is similar to `let`:

Listing 12.30: Use binding expression.

```
use <valueIdent> = <bodyExpr> [in <expr>]
```

A `use` binding provides a binding between the `<bodyExpr>` expression to the name `<valueIdent>` in the following expression(s), and in contrast to `let`, it also adds a call to `Dispose()` on `<valueIdent>` if it implements `System.IDisposable`. See for example Listing 12.31. Here, `file` is an `System.IDisposable` object,

Listing 12.31 useBinding.fsx:
Using `use` instead of `let` releases disposable resources at end of scope.

```
open System.IO

let writeToFile (filename : string) (str : string) : unit =
   use file = File.CreateText filename
   file.Write str
   // file.Dispose() is implicitly called here,
   // implying that the file is closed.

writeToFile "use.txt" "'Use' cleans up, when out of scope."
```

and `file.Dispose()` is called automatically before `writeToFile` returns. This implies that the file is closed. Had we used `let` instead, then the file would first be closed when the program terminates.

The higher-order function *`using`* takes a disposable object and a function, executes the function on the disposable objects, and then calls `Dispose()` on the disposable object. This is illustrated in Listing 12.32

Listing 12.32 using.fsx:
The `using` function executes a function on an object and releases its disposable resources. Compare with Listing 12.31.

```
open System.IO

let writeToFile (str : string) (file : StreamWriter) : unit =
    file.Write str

using (File.CreateText "use.txt") (writeToFile "Disposed
    after call.")
// Dispose() is implicitly called on the anonymous file
// handle, implying that the file is automatically closed.
```

The main difference between `use` and `using` is that resources allocated using `use` are disposed at the end of its scope, while `using` disposes the resources after the execution of the function in its argument. In spite of the added control of `using`, we

★ **prefer `use` over `using` due to its simpler structure.**

12.8 Key concepts and terms in this chapter

In this chapter, we have looked at how to interface programs with external data sources. Key concepts have been:

- **Arguments to programs** run in the terminal can be retrieved from within a program at running time
- A **file** is a general concept, for places to put data to or retrieve data from.
- Files are often a shared resource, and there may be other programs on a machine, which need access.
- To avoid chaos, the operating system **locks access to files** by other programs and therefore it is also important to **release the locks**, when your program is done.
- A **stream** is similar to a file for reading, however, they do not allow for random access.

- Errors may occur due to events outside the domain of control for a program. Such errors give rise to **exceptions**.
- Exceptions may be **caught** or **thrown** by our programs.

Chapter 13

Imperative programming

Abstract Imperative programming is both and overarching term for a collection of programming paradigms and the name for specific paradigm. Here we will discuss the specific paradigm, and in Chapters 15 to 17 we will discuss another imperative paradigm called the object-oriented paradigm. When programming using the imperative paradigm, the programmer specifies a number of steps which to maninipulate the state of the computer for a desired result. Key features of this paradigm are *mutable values* also know as *variables* and `for`- and `while`-loops. In this chapter, you will learn

- How to define and use variables
- How to make loops using the `for`- and `while`-keywords as an alternative to recursive functions
- How to arrays as alternative to lists
- How to trace-by-hand imperative programs

13.1 Variables

A state is a value that may change over time. E.g., a traffic light as shown in Figure 13.1, consists of 3 colored lamps: red (top), yellow (middle), and green (bottom), and typically cycles through the cyclic sequence red, red+yellow, green, yellow. A simple model of this could be to represent the state of each lamp as being

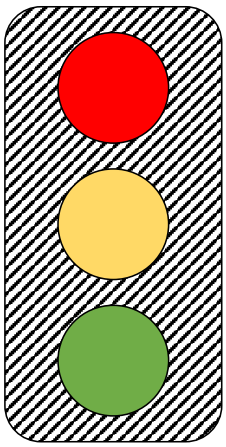


Fig. 13.1 For a traffic light, the different state of the coloured lamps can be modelled as different states of the ligth. Top lamp is red, middle is yellow, and bottom is green.

on or off. This can be done with a mutable boolean value. To create a mutable value, we initally declare and identifyer using the `mutable` keyword with the following syntax:

Listing 13.1: Syntax for defining mutable values with an initial value.

```
1 let mutable <ident> = <expr>
```

Changing the value of an identifier is called *assignment* and is done using the "<-" lexeme. Assignments have the following syntax:

Listing 13.2: Value reassignment for mutable variables.

```
1 <ident> <- <expr>
```

Mutable values is synonymous with the term *variable*. A variable is an area in the computer's working memory associated with an identifier and a type, and this area may be read from and written to during program execution, see Listing 13.3 for an example. Here, an area in memory was denoted `x`, initially assigned the integer value 5, hence the type was inferred to be `int`. Later, this value of `x` was replaced with another integer using the "<-" lexeme. The "<-" lexeme is used to distinguish

Listing 13.3 mutableAssignReassingShort.fsx: A variable is defined and later reassigned a new value.

```
let mutable x = 5
printfn "%d" x
x <- -3
printfn "%d" x
```

```
$ dotnet fsi mutableAssignReassingShort.fsx
5
-3
```

the assignment from the comparison operator. For example, the statement `a = 3` in Listing 13.4 is not an assignment but a comparison which is evaluated to be false.

Listing 13.4: It is a common error to mistake "=" and "<-" lexemes for mutable variables.

```

> let mutable a = 0
a = 3;;
val mutable a: int = 0
val it: bool = false
```

Assignment type mismatches will result in an error, as demonstrated in Listing 13.5. I.e., once the type of an identifier has been declared or inferred, it cannot be changed.

Listing 13.5 mutableAssignReassingTypeError.fsx: Assignment type mismatching causes a compile-time error.

```
let mutable x = 5
printfn "%d" x
x <- -3.0
printfn "%d" x
```

```
$ dotnet fsi mutableAssignReassingTypeError.fsx


mutableAssignReassingTypeError.fsx(3,6): error FS0001: This
   expression was expected to have type
    'int'
but here has type
    'float'
```

A typical variable is a counter of type integer, and a typical use of counters is to increment them, see Listing 13.6 for an example. Using variables in expressions, as opposed to the left-hand-side of an assignment operation, reads the value of the variable. Thus, when using a variable as the return value of a function, then the value

Listing 13.6 mutableAssignIncrement.fsx:
Variable increment is a common use of variables.

```
let mutable x = 5 // Declare a variable x and assign the
   value 5 to it
printfn "%d" x
x <- x + 1 // Increment the value of x
printfn "%d" x
```

```
$ dotnet fsi mutableAssignIncrement.fsx
5
6
```

is copied from the local scope of the function to the scope from which it is called. This is demonstrated in Listing 13.7. In the example, we see that the type is a value,

Listing 13.7: Returning a mutable variable returns its value.

```
> let g () =
  let mutable y = 0
  y
let y' = g();;
val g: unit -> int
val y': int = 0
```

and not mutable.

In F# it is possible to encapsulate a mutable value. For example, consider a counter function `inc: unit->int`, which increments a counting state and returns its present value, every time we call it, i.e., calling `inc ()` the first time should return the value 1, the second time the value 2, and so on. A first attempt could be as shown in Listing 13.8. Even though `inc` has an internal state, the identifier "`s`" is reset

Listing 13.8 inc.fsx:
A failed version of the a counter function `inc`.

```
let inc () =
  let mutable s = 0
  s <- s + 1
  s

printfn "%A" (inc ())
printfn "%A" (inc ())
printfn "%A" (inc ())
```

```
$ dotnet fsi inc.fsx
1
1
1
```

to 0 every time the function is called. However, in F# it is possible to solve this problem by using a side-effect as shown in Listing 13.9. The reason this works is

Listing 13.9 makeCounter.fsx:
Returning a counter function with a side-effect. Compare with Listing 13.8.

```
let makeCounter () =
  let mutable s = 0
  fun () ->
    s <- s + 1
    s

let inc = makeCounter ()
printfn "%A" (inc ())
printfn "%A" (inc ())
printfn "%A" (inc ())
```

```
$ dotnet fsi makeCounter.fsx
1
2
3
```

that `makeCounter` returns a closure that includes the environment of the anonymous function at the point where it is defined. Hence, this closure includes a mutable value but does not reset it, every time it is called.

Variables implement dynamic scope, that is, the value of an identifier depends on *when* it is used. This is in contrast to lexical scope, where the value of an identifier depends on *where* it is defined. As an example, consider the script in Listing 4.16 which defines a function using lexical scope and returns the number `6.0`, however, if `a` is made `mutable`, then the behavior is different, as shown in Listing 13.10. Here,

Listing 13.10 dynamicScopeNFunction.fsx:
Mutual variables implement dynamic scope rules. Compare with Listing 4.16.

```
let testScope x =
  let mutable a = 3.0
  let f z = a * z
  a <- 4.0
  f x
printfn "%A" (testScope 2.0)
```

```
$ dotnet fsi dynamicScopeNFunction.fsx
8.0
```

the response is `8.0`, since the value of `a` changed before the function `f` was called.

13.2 While and For Loops

This book has previously emphasized recursion as a structure to encapsulate and repeat code. In the imperative paradigm, often *for*- and *while*-loops are preferred. A *while*-loop has the following syntax:

Listing 13.11: While loop.

```
while <condition> do
  <expr>
```

The *condition* <condition> is an expression that evaluates to true or false. A while-loop repeats the <expr> expression as long as the condition is true. The *do* keyword is mandatory and the body of the loop is indicated by indentation as usual. The return value of the while expression is "()".

The program in Listing 13.14 is an example of a while-loop which counts from 1 to 10. Since the variable i is used for counting, it is often called the counter variable.

Listing 13.12 countWhile.fsx:
Count to 10 with a counter variable.

```
let mutable i = 1
while i <= 10 do
  printf "%d " i
  i <- i + 1
printf "\n"
```

```
$ dotnet fsi countWhile.fsx
1 2 3 4 5 6 7 8 9 10
```

The counting is done by performing the following computation: In line 1, the counter variable is first given an initial value of 1. Then in line 2, the head of the while-loop and examines the condition. Since 1 <= 10, the condition is true, and execution enters the body of the loop starting in line 3. The body prints the value of the counter to the screen and increases the counter by 1. Then execution returns to the head of the while-loop and reexamines the condition. Now the condition is 2 <= 10, which is also true, and so execution enters the body and so on until the counter has reached the value 11, in which case the condition 11 <= 10 is false, and execution continues in line 5.

Counters are so common that a special syntax has been reserved for loops using counters. These are called *for*-*to*-loops. For-loops come in several variants, and here we will focus on the one using an explicit counter. Its syntax is:

Listing 13.13: For loop.

```
for <ident> = <firstExpr> to <lastExpr> do
  <expr>
```

A for-loop initially binds the counter identifier `<ident>` to be the value `<firstExpr>`. Then execution enters the body, and `<bodyExpr>` is evaluated. Once done, the counter is increased, and execution evaluates `<bodyExpr>` once again. This is repeated as long as the counter is not greater than `<lastExpr>`. Againg, the *do* keyword is mandatory and the body of the loop is indicated by indentation as usual. The return value of the `for` expression is "`()`".

The counting example from Listing 13.12 using a `for`-loop is shown in Listing 13.14 As this interactive script demonstrates, the identifier `i` takes all the values between

Listing 13.14 count.fsx:
Counting from 1 to 10 using a `for`-loop.

```
for i = 1 to 10 do printf "%d " i done
printfn ""
```

```
$ dotnet fsi count.fsx
1 2 3 4 5 6 7 8 9 10
```

1 and 10, but in spite of its changing state, it is not mutable.

Counting backwards is sufficiently common that F# has a *for-downto* structure, which works exactly like a `for`-`to`-loop except that the counter is decreased by 1 in each iteration. An example of this is shown in Listing 13.15.

Listing 13.15 countBackwards.fsx:
Counting from 10 to 1 using a `for`-`downto`-loop.

```
for i = 10 downto 1 do
  printf "%d " i
printfn ""
```

```
$ dotnet fsi countBackwards.fsx
10 9 8 7 6 5 4 3 2 1
```

There is also a customized syntax for indexing lists as shown in Listing 13.16 This particular syntax for sequentially indexing into lists using a for loop is to be prefered, since it completely avoids index-out-of-bound errors.

Listing 13.16 listFor.fsx:
Iterating over elements of a list with the `for`-`in`-loop.

```
for elm in [3..2..9] do
  printf "%A " elm
printfn ""
```

```
$ dotnet fsi listFor.fsx
3 5 7 9
```

13.3 Programming Intermezzo: Imperative Fibonacci numbers

To further compare for- and while-loops, consider the following problem.

Problem 13.1
Write a program that calculates the n'th Fibonacci number.

Fibonacci numbers is a sequence of numbers starting with 1, 1, and where the next number is calculated as the sum of the previous two. Hence the first ten numbers are: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55. Fibonacci numbers are related to Golden spirals shown in Figure 13.2. Often the sequence is extended with a preceding number 0, to be

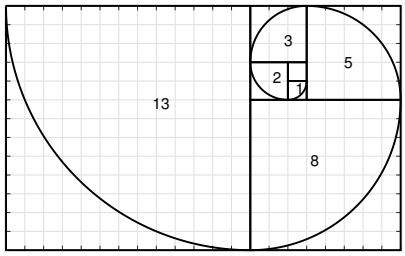


Fig. 13.2 The Fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive Fibonacci numbers, and the curve in each square is the circular arc with a radius of the square it is drawn in.

0, 1, 1, 2, 3, . . . , which we will do here as well.

In Listing 8.5 we gave a solution using recursion. Here we will first use a `for`-loop, as shown in Listing 13.17. The basic idea of the solution is that if we are given the $(n-1)$'th and $(n-2)$'th numbers, the n'th number is trivial to compute. And assuming that fib(1) and fib(2) are given, then it is trivial to calculate fib(3). For fib(4), we only need fib(3) and fib(2), hence we may disregard fib(1). Thus, we realize that we can cyclicly update the previous, current, and next values by shifting values until we have reached the desired fib(n). This is implement in Listing 13.17 as the function `fib`, which takes an integer `n` as argument and returns the n'th Fibonacci number. The function does this iteratively using a `for`-loop, where `i` is the counter value, and `pair` is the pair of the $i-1$'th and i'th Fibonacci numbers. In the body of the

Listing 13.17 fibFor.fsx:
The *n*'th Fibonacci number calculated using a for-loop.

```
let fib n =
  let mutable pair = (0, 1)
  for i = 2 to n do
    pair <- (snd pair, (fst pair) + (snd pair))
  snd pair

printfn "fib(1) = %d" (fib 1)
printfn "fib(2) = %d" (fib 2)
printfn "fib(3) = %d" (fib 3)
printfn "fib(10) = %d" (fib 10)
```

```
$ dotnet fsi fibFor.fsx
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(10) = 55
```

loop, the i'th and $i+1$'th numbers are assigned to `pair`. The `for`-loop automatically updates `i` for next iteration. When $n < 2$ the body of the for-loop is not evaluated, and 1 is returned. This is of course wrong for $n < 1$, but we will ignore this for now.

Listing 13.18 shows a program similar to Listing 13.17 using a while-loop instead of for-loop. The programs are almost identical. In this case, the `for`-loop is to

Listing 13.18 fibWhile.fsx:
The *n*'th Fibonacci number calculated using a while-loop.

```
let fib (n : int) : int =
  let mutable pair = (0, 1)
  let mutable i = 1
  while i < n do
    pair <- (snd pair, fst pair + snd pair)
    i <- i + 1
  snd pair

printfn "fib(1) = %d" (fib 1)
printfn "fib(2) = %d" (fib 2)
printfn "fib(3) = %d" (fib 3)
printfn "fib(10) = %d" (fib 10)
```

```
$ dotnet fsi fibWhile.fsx
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(10) = 55
```

be preferred, since more lines of code typically mean more chances of making a mistake. However, while-loops are somewhat easier to argue correctness about.

While-loops also allow for logical structures other than for-loops, such as the case when the number of iteration cannot easily be decided when entering the loop. As an example, consider a slight variation of the above problem, where we wish to find the largest Fibonacci number less or equal some number. A solution to this problem is shown in Listing 13.19. The strategy here is to iteratively calculate Fibonacci

Listing 13.19 fibWhileLargest.fsx:
Search for the largest Fibonacci number less than a specified number.

```
let largestFibLeq n =
  let mutable pair = (0, 1)
  while snd pair <= n do
    pair <- (snd pair, fst pair + snd pair)
  fst pair

for i = 1 to 10 do
  printfn "largestFibLeq(%d) = %d" i (largestFibLeq i)
```

```
$ dotnet fsi fibWhileLargest.fsx
largestFibLeq(1) = 1
largestFibLeq(2) = 2
largestFibLeq(3) = 3
largestFibLeq(4) = 3
largestFibLeq(5) = 5
largestFibLeq(6) = 5
largestFibLeq(7) = 5
largestFibLeq(8) = 8
largestFibLeq(9) = 8
largestFibLeq(10) = 8
```

numbers until we've found one larger than the argument `n`, and then return the previous. This could not be calculated with a for-loop.

13.4 Arrays

One dimensional *arrays*, or just arrays for short, are mutable lists of the same type and follow a similar syntax as lists. Arrays can be stated as a *sequence expression*,

Listing 13.20: The syntax for an array using the sequence expression.

```
[|[<expr>{; <expr>}]|]
```

E.g., `[|1; 2; 3|]` is an array of integers, `[|"This"; "is"; "an"; "array"|]` is an array of strings, `[|(fun x -> x); (fun x -> x*x)|]` is an array of functions, `[||]` is the empty array. Arrays may also be given as ranges,

Listing 13.21: The syntax for an array using the range expression.

```
[|<expr> .. <expr> [.. <expr>]|]
```

but arrays of *range expressions* must be of `<expr>` integers, floats, or characters. Examples are `[|1 .. 5|]`, `[|-3.0 .. 2.0|]`, and `[|'a' .. 'z'|]`. Range expressions may include a step size, thus, `[|1 .. 2 .. 10|]` evaluates to `[|1; 3; 5; 7; 9|]`.

The array type is defined using the `array` keyword or alternatively the "`[]`" lexeme. Like strings and lists, arrays may be indexed using the *"`[]`"* notation. Arrays cannot be resized, but are mutable, as shown in Listing 13.22. Notice that in spite of

Listing 13.22 arrayReassign.fsx:
Arrays are mutable in spite of the missing `mutable` keyword.

```
let square (a : int array) =
  for i = 0 to a.Length - 1 do
    a[i] <- a[i] * a[i]

let A = [| 1; 2; 3; 4; 5 |]
printfn "%A" A
square A
printfn "%A" A
```

```
$ dotnet fsi arrayReassign.fsx
[|1; 2; 3; 4; 5|]
[|1; 4; 9; 16; 25|]
```

the missing `mutable` keyword, the function `square` still has the *side-effect* of squaring all entries in `A`. F# implements arrays as chunks of memory and indexes arrays via address arithmetic. I.e., element i in an array, whose first element is in memory address α and whose elements fill β addresses each, is found at address $\alpha + i\beta$. Hence, indexing has computational complexity of $O(1)$, but appending and prepending values to arrays and array concatenation requires copying the new and existing values to a fresh area in memory and thus has computational complexity $O(n)$, where n is the total number of elements. Thus, **indexing arrays is fast, but cons and concatenation is slow and should be avoided.** ⋆

Arrays support *slicing*, that is, indexing an array with a range result in a copy of the array with values corresponding to the range. This is demonstrated in Listing 13.23. As illustrated, the missing start or end index imply from the first or to the last element, respectively.

Listing 13.23: Examples of array slicing. Compare with Listing 7.3 and Listing 3.27.

```
> let arr = [|'a' .. 'g'|];;
val arr: char[] = [|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'|]

> arr[0];;
val it: char = 'a'

> arr[3];;
val it: char = 'd'

> arr[3..];;
val it: char[] = [|'d'; 'e'; 'f'; 'g'|]

> arr[..3];;
val it: char[] = [|'a'; 'b'; 'c'; 'd'|]

> arr[1..3];;
val it: char[] = [|'b'; 'c'; 'd'|]

> arr[*];;
val it: char[] = [|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'|]
```

Arrays do not have explicit operator support for appending and concatenation, instead the `Array` namespace includes an `Array.append` function, as shown in Listing 13.24.

Listing 13.24 arrayAppend.fsx: Two arrays are appended with `Array.append`.

```
let a = [|1; 2;|]
let b = [|3; 4; 5|]
let c = Array.append a b
printfn "%A, %A, %A" a b c
```

```
$ dotnet fsi arrayAppend.fsx
[|1; 2|], [|3; 4; 5|], [|1; 2; 3; 4; 5|]
```

Arrays are *reference types*, meaning that identifiers are references and thus suffer from aliasing, as illustrated in Listing 13.25.

Listing 13.25 arrayAliasing.fsx:
Arrays are reference types and suffer from aliasing.

```
let a = [|1; 2; 3|];
let b = a
a[0] <- 0
printfn "a = %A, b = %A" a b;;
```

```
$ dotnet fsi arrayAliasing.fsx
a = [|0; 2; 3|], b = [|0; 2; 3|]
```

13.4.1 Array Properties and Methods

Some important properties and methods for arrays are:

`Clone(): 'T []`.
Returns a copy of the array.

Listing 13.26: `Clone`

```
> let a = [|1; 2; 3|];
let b = a.Clone()
a[0] <- 0
printfn "a = %A, b = %A" a b;;
a = [|0; 2; 3|], b = [|1; 2; 3|]
val a: int[] = [|0; 2; 3|]
val b: obj = [|1; 2; 3|]
val it: unit = ()
```

`Length: int`.
Returns the number of elements in the array.

Listing 13.27: `Length`

```
> [|1; 2; 3|].Length;;
val it: int = 3
```

13.4.2 The Array Module

There are quite a number of built-in procedures for arrays in the `Array` module, and many of them are almost identical to those in the `List` module, discussed in Section 7.1. However, a few additional functions are noted below:

`Array.append: arr1:'T [] -> arr2:'T [] -> 'T []`.
Creates an new array whose elements are a concatenated copy of `arr1` and `arr2`.

Listing 13.28: `Array.append`

```
> Array.append [|1; 2;|] [|3; 4; 5|];;
val it: int[] = [|1; 2; 3; 4; 5|]
```

`Array.copy: 'T [] -> 'T []`.
Creates an array that contains the elements of the supplied array.

Listing 13.29: `Array.copy`

```
> let a = [|1; 2; 3|]
let b = Array.copy a;;
val a: int[] = [|1; 2; 3|]
val b: int[] = [|1; 2; 3|]
```

`Array.ofList`: `lst:'T list -> 'T []`.
Creates an array whose elements are copied from `lst`.

Listing 13.30: `Array.ofList`

```
> Array.ofList [1; 2; 3];;
val it: int[] = [|1; 2; 3|]
```

`Array.toList`: `arr:'T [] -> 'T list`.
Creates a new list whose elements are copied from `arr`.

Listing 13.31: `Array.toList`

```
> Array.toList [|1; 2; 3|];;
val it: int list = [1; 2; 3]
```

13.4.3 Multidimensional Arrays

Multidimensional arrays can be created as arrays of arrays (of arrays . . .). These are known as *jagged arrays*, since there is no inherent guarantee that all sub-arrays are of the same size. The example in Listing 13.32 is a jagged array of increasing width. Indexing arrays of arrays is done sequentially, in the sense that in the above example,

Listing 13.32 arrayJagged.fsx:
An array of arrays of non-equal lenghts is a jagged array.

```
let arr = [|[|1|]; [|1; 2|]; [|1; 2; 3|]|]

for row in arr do
  for elm in row do
    printf "%A " elm
  printf "\n"
```

```
$ dotnet fsi arrayJagged.fsx
1
1 2
1 2 3
```

the number of outer arrays is `a.Length`, `a[i]` is the i'th array, the length of the

i'th array is `a[i].Length`, and the j'th element of the i'th array is thus `a[i][j]`. Often 2-dimensional rectangular arrays are used, which can be implemented as a jagged array, as shown in Listing 13.33. Note that, the `arr[i][j]` argument in

Listing 13.33 arrayJaggedSquare.fsx: A rectangular array.

```
1  let pownArray  (arr : int array array) p =
2    for i = 1 to arr.Length - 1 do
3      for j = 1 to arr[i].Length - 1 do
4        arr[i][j] <- pown (arr[i][j]) p
5
6  let printArrayOfArrays (arr : int array array) =
7    for row in arr do
8      for elm in row do
9        printf "%3d " elm
10     printf "\n"
11
12 let A = [|[|1 .. 4|]; [|1 .. 2 .. 7|]; [|1 .. 3 .. 10|]|]
13 pownArray A 2
14 printArrayOfArrays A
```

```
1  $ dotnet fsi arrayJaggedSquare.fsx
2    1   2   3   4
3    1   9  25  49
4    1  16  49 100
```

line 4 must be parenthesized to avoid ambiguity. Further, the `for-in` cannot be used in `pownArray`, e.g.,

```
for row in arr do
  for elm in row do
    elm <- pown elm p
```

since the iterator value `elm` is not mutable, even though `arr` is an array.

Square arrays of dimensions 2 to 4 are so common that F# has built-in modules for their support. Here, we will describe *`Array2D`*. The workings of *`Array3D`* and *`Array4D`* are very similar. A generic `Array2D` has type `'T [,]`, and it is indexed also using the `[,]` notation. The `Array2D.length1` and `Array2D.length2` functions are supplied by the `Array2D` module for obtaining the size of an array along the first and second dimension. Rewriting the with jagged array example in Listing 13.33 to use `Array2D` gives a slightly simpler program, which is shown in Listing 13.34. Note that the `printf` supports direct printing of the 2-dimensional array. `Array2D` arrays support slicing. The "*" lexeme is particularly useful to obtain all values along a dimension. This is demonstrated in Listing 13.35. Note that in almost all cases, slicing produces a sub-rectangular 2 dimensional array, except for `arr[1,*]`, which is an array, as can be seen by the single "[". In contrast, `A[1..1,*]` is an

Listing 13.34 array2D.fsx:
Creating a 3 by 4 rectangular array of integers.

```
let arr = Array2D.create 3 4 0
for i = 0 to (Array2D.length1 arr) - 1 do
  for j = 0 to (Array2D.length2 arr) - 1 do
    arr[i,j] <- j * Array2D.length1 arr + i
printfn "%A" arr
```

```
$ dotnet fsi array2D.fsx
[[0; 3; 6; 9]
 [1; 4; 7; 10]
 [2; 5; 8; 11]]
```

Listing 13.35: Examples of Array2D slicing. Compare with Listing 13.34.

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j);;
val arr: int[,] = [[0; 10; 20; 30]
                   [1; 11; 21; 31]
                   [2; 12; 22; 32]]

> arr[2,3];;
val it: int = 32

> arr[1..,3..];;
val it: int[,] = [[31]
                  [32]]

> arr[..1,*];;
val it: int[,] = [[0; 10; 20; 30]
                  [1; 11; 21; 31]]

> arr[1,*];;
val it: int[] = [|1; 11; 21; 31|]

> arr[1..1,*];;
val it: int[,] = [[1; 11; 21; 31]]
```

Array2D. Note also that `printfn` typesets 2 dimensional arrays as `[[ ... ]]` and not `[|[| ... |]|]`, which can cause confusion with lists of lists.

Multidimensional arrays have the same properties and methods as arrays, see Section 13.4.1.

13.4.4 The Array2D Module

The `Array2D` module is somewhat limited. In particular neither `fold` nor `foldback` functions exists. Most of the module's functions are mentioned below:

`copy: arr:'T [,] -> 'T [,].`
Creates a new array whose elements are copied from `arr`.

Listing 13.36: `Array2D.copy`

```
> let a = Array2D.init 3 4 (fun i j -> i + 10 * j)
let b = Array2D.copy a;;
val a: int[,] = [[0; 10; 20; 30]
                 [1; 11; 21; 31]
                 [2; 12; 22; 32]]
val b: int[,] = [[0; 10; 20; 30]
                 [1; 11; 21; 31]
                 [2; 12; 22; 32]]
```

`create: m:int -> n:int -> v:'T -> 'T [,].`
Creates an `m` by `n` array whose elements are set to `v`.

Listing 13.37: `Array2D.create`

```
> Array2D.create 2 3 3.14;;
val it: float[,] = [[3.14; 3.14; 3.14]
                    [3.14; 3.14; 3.14]]
```

`init: m:int -> n:int -> f:(int -> int -> 'T) -> 'T [,].`
Creates an `m` by `n` array whose elements are the result of applying `f` to the index of an element.

Listing 13.38: `Array2D.init`

```
> Array2D.init 3 4 (fun i j -> i + 10 * j);;
val it: int[,] = [[0; 10; 20; 30]
                  [1; 11; 21; 31]
                  [2; 12; 22; 32]]
```

`iter: f:('T -> unit) -> arr:'T [,] -> unit.`
Applies `f` to each element of `arr`.

Listing 13.39: `Array2D.iter`

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j)
Array2D.iter (fun elm -> printf "%A " elm) arr
printfn "";;
0 10 20 30 1 11 21 31 2 12 22 32
val arr: int[,] = [[0; 10; 20; 30]
                   [1; 11; 21; 31]
                   [2; 12; 22; 32]]
val it: unit = ()
```

`length1: arr:'T [,] -> int`.
Returns the length the first dimension of `arr`.

Listing 13.40: `Array2D.length1`

```
> let arr = Array2D.create 2 3 0.0 in Array2D.length1
   arr;;
val it: int = 2
```

`length2: arr:'T [,] -> int`.
Returns the length of the second dimension of `arr`.

Listing 13.41: `Array2D.forall length2`

```
> let arr = Array2D.create 2 3 0.0 in Array2D.length2
   arr;;
val it: int = 3
```

`map: f:('T -> 'U) -> arr:'T [,] -> 'U [,]`.
Creates a new array whose elements are the results of applying `f` to each of the elements of `arr`.

Listing 13.42: `Array2D.map`

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j)
Array2D.map (fun x -> x * x) arr;;
val arr: int[,] = [[0; 10; 20; 30]
                   [1; 11; 21; 31]
                   [2; 12; 22; 32]]
val it: int[,] = [[0; 100; 400; 900]
                  [1; 121; 441; 961]
                  [4; 144; 484; 1024]]
```

13.5 Tracing Imperative Programs

Debugging imperative programs is more complicated than declarative programs. In particular, the notion of states require us to keep track of the dynamic scope of values.

In the following, we will discuss trace-by-hand of `for`-loops followed by programs which involves states.

13.5.1 Tracing Loops

Consider the program in Listing 13.43. The program includes a function for printing

Listing 13.43 printSquares.fsx:
Print the squares of a sequence of positive integers.

```
1  let N = 3
2  let printSquares n =
3    for i = 1 to n do
4      let p = i * i
5      printfn "%d: %d" i p
6
7  printSquares N
```

```
$ dotnet fsi printSquares.fsx
1: 1
2: 4
3: 9
```

the sequence of the first N squares of integers. It uses a `for`-loop with a counting value. F# creates a new environment each time the loop body is executed. Thus, to trace this program, we mentally *unfold* the loop as shown in Listing 13.44. The unfolding contains 3 new scopes lines 3–7, lines 8–12, and lines 13–17 corresponding to the 3 times, the loop is repeated, and each scope starts by binding the counting value to the name `i`.

In the rest of this section, we will refer to the code in Listing 13.43. The first rows in our tracing-table looks as follows:

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	$N = 3$
2	2	E_0	printSquares = $\big((n)$, printSquares-body, $(N = 3)\big)$
3	7	E_0	printSquares N = ?

Note that due to the lexical scope rule, the closure of `printSquares` includes `N` in its environment element. Calling `printSquares N` causes the creation of a new environment,

Listing 13.44 printSquaresUnfold.fsx: An unfolded version of Listing 13.43.

```
let N = 3
let printSquaresUnfold n =
  (
    let i = 1
    let p = i * i
    printfn "%d: %d" i p
  )
  (
    let i = 2
    let p = i * i
    printfn "%d: %d" i p
  )
  (
    let i = 3
    let p = i * i
    printfn "%d: %d" i p
  )

printSquaresUnfold N
```

```
$ dotnet fsi printSquaresUnfold.fsx
1: 1
2: 4
3: 9
```

Step	Line	Env.	Bindings and evaluations
4	2	E_1	$((n = 3), \text{printSquares-body}, (N = 3))$

The first statement of printSquares-body is the `for`-loop. As our unfolding in Listing 13.44 demonstrated, each time the loop-body is executed, a new scope is created with a new binding to `i`. Reusing the notation from closures, we write

Step	Line	Env.	Bindings and evaluations
5	3	E_1	$\text{for} \cdots = ?$

and create a new environment as if it had been a function,

Step	Line	Env.	Bindings and evaluations
6	3	E_2	$((i = 1), \text{for-body}, (n = 3, N = 3))$

As for functions, this denotes the bindings available at beginning of the execution of the `for`-body. The first line in the `for`-body is the binding of the value of an expression to `p`. The expression is `i*i`, and to calculate its value, we look in the `for`-loop's pseudo-closure where we find the $i = 1$ binding. Hence,

Step	Line	Env.	Bindings and evaluations
7	4	E_2	$i * i = 1$
8	4	E_2	$p = 1$

The final step in the for-body is the `printfn`-statement. Its arguments we get from the updated, active environment E_2 and write,

Step	Line	Env.	Bindings and evaluations
9	5	E_2	output = "1 : 1\n"

At this point, the `for`-loop has reached its last line, E_2 is deleted, we create a new environment with the counter variable increased by 1, and repeat. Hence,

Step	Line	Env.	Bindings and evaluations
10	3	E_3	$\big((i = 2)$, for-body, $(n = 3, N = 3)\big)$
11	4	E_3	$i * i = 4$
12	4	E_3	$p = 4$
13	5	E_3	output = "2 : 4\n"

Again, we delete E_3, create E_4 where i is incremented, and repeat,

Step	Line	Env.	Bindings and evaluations
14	3	E_4	$\big((i = 3)$, for-body, $(n = 3, N = 3)\big)$
15	4	E_4	$i * i = 9$
16	4	E_4	$p = 9$
17	5	E_4	output = "3 : 9\n"

Finally, incrementing i would mean that $i > n$, hence the `for`-loop ends and as all statements returns `()`

Step	Line	Env.	Bindings and evaluations
18	3	E_4	return = ()

At this point, the environment E_4 is deleted, and we return to the enclosing environment E_1 and the statement or expression following Step 5. Since the `for`-loop is the last expression in the `printSquares` function, its return value is that of the `for`-loop,

Step	Line	Env.	Bindings and evaluations
19	3	E_1	return = ()

Returning to Step 3 and environment E_0, we have now calculated the return-value of `printSquares N` to be `()`, and since this line is the last of our program, we return `()` and end the program:

Step	Line	Env.	Bindings and evaluations
20	3	E_0	return = ()

13.5.2 Tracing Mutable Values

For mutable bindings, the scope is dynamic. For this, we need the concept of storage. Consider the program in Listing 13.45. To trace the dynamic behavior of this

Listing 13.45 dynamicScopeTracing.fsx:
Example of lexical scope and closure environment.

```
let testScope x =
  let mutable a = 3.0
  let f z = a * z
  a <- 4.0
  f x
printfn "%A" (testScope 2.0)
```

```
$ dotnet fsi dynamicScopeTracing.fsx
8.0
```

program, we add a second table to our hand tracing, which is initially empty and has the columns Step and Value to hold the Step number when the value was updated and the value stored. For Listing 13.45, the firsts 4 steps thus look like,

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	testScope = $\big((x), \text{body}, ()\big)$
2	6	E_0	testScope 2.0 = ?
3	1	E_1	$\big((x = 2.0), \text{body}, ()\big)$

Step	Value
0	-

The mutable binding in line 2 creates an internal name and a dynamic storage location. The name `a` will be bound to a reference value, which we call α_1, and which is a unique name shared between the two tables:

Step	Line	Env.	Bindings and evaluations
4	2	E_1	$a = \alpha_1$

Step	Value
4	$\alpha_1 = 3.0$

The following closure of `f` uses the reference-name instead of its value,

Step	Line	Env.	Bindings and evaluations
5	3	E_1	$f = \big((z), a * z, (x = 2.0, a = \alpha_1)\big)$

Step	Value
4	$\alpha_1 = 3.0$

In line 4, the value in the storage is updated by the assignment operator, which we denote as,

Step	Line	Env.	Bindings and evaluations
6	4	E_1	a <- 4.0

Step	Value
6	$\alpha_1 = 4.0$

Hence, when we evaluate the function `f`, its closure looks up the value of `a` by following the reference and finding the new value:

Step	Line	Env.	Bindings and evaluations
7	5	E_1	f x = ?
8	5	E_2	$\big((z = 2.0), a * z, (x = 2.0, a = \alpha_1)\big)$
9	5	E_2	$a * z = 8.0$
10	5	E_2	return = 8.0
10	5	E_1	return = 8.0
11	6	E_0	output = "8.0\n"
12	6	E_0	return = ()

Step	Value
6	$\alpha_1 = 4.0$

For reference, the complete pair of tables is shown in Table 13.1. By comparing this

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	testScope = $\big((x), \text{body}, ()\big)$
2	6	E_0	testScope 2.0 = ?
3	1	E_1	$\big((x = 2.0), \text{body}, ()\big)$
4	2	E_1	$a = \alpha_1$
5	3	E_1	$f = \big((z), a * z, (x = 2.0, a = \alpha_1)\big)$
6	4	E_1	a <- 4.0
7	5	E_1	f x = ?
8	5	E_2	$\big((z = 2.0), a * z, (x = 2.0, a = \alpha_1)\big)$
9	5	E_2	$a * z = 8.0$
10	5	E_2	return = 8.0
10	5	E_1	return = 8.0
11	6	E_0	output = "8.0\n"
12	6	E_0	return = ()

Step	Value
0	-
4	$\alpha_1 = 3.0$
6	$\alpha_1 = 4.0$

Table 13.1 The complete table produced while tracing the program in Listing 13.45 by hand.

to the value-bindings in Listing 4.32, we see that the mutable values give rise to a different result due to the difference between lexical and dynamic scope.

13.6 Key Concepts and Terms in This Chapter

In this chapter, we have looked at programming with states using the imperative programming paradigm. You have seen how to:

- define **mutable variables** and make loops with **while**- and **for**- loops
- work with **arrays**, **jagged arrays**, and **2-dimensional arrays**.
- **trace-by-hand** programs involving mutable values and for-loops.

Chapter 14

Testing Programs

Abstract When programming, chances are that errors or bugs are created. Some are syntactical, which dotnet is very good at finding, but others are semantical, which can be very hard to find. To systematically seek semantical bugs, you will in this chapter learn how to

- test sofware without from the persective of a user or a customer, who has no knowledge about the internal structure of the software, but who is interested in its functionality. This is known as black-box testing
- test software from a perspective of the software developer, with full knowledge and with a focus on every line of code. This is of known as white-box testing

A software bug is an error in a computer program that causes it to produce an incorrect result or behave in an unintended manner. The term 'bug' was used by Thomas Edison in 1878[1][2], but made popular in computer science by Grace Hopper, who found a moth interfering with the electronic circuits of the Harward Mark II electromechanical computer and coined the term *bug* for errors in computer programs. The original bug is shown in Figure 14.1. Software is everywhere, and

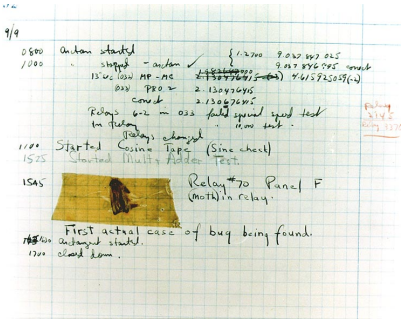


Fig. 14.1 The first computer bug, caught by Grace Hopper, U.S. Naval Historical Center Online Library Photograph NH 96566-KN.

errors therein have a huge economic impact on our society and can threaten lives[3].

The ISO/IEC organizations have developed standards for software testing[4]. To illustrate basic concepts of software quality, consider a hypothetical route planning system. Essential factors of its quality are:

Functionality: Does the software compile and run without internal errors. Does it solve the problem it was intended to solve? E.g., does the route planning software find a suitable route from point a to b?

Reliability: Does the software work reliably over time? E.g., does the route planning software work when there are internet dropouts?

Usability: Is the software easy and intuitive to use by humans? E.g., is it easy to enter addresses and alternative routes in the software's interface?

Efficiency: How many computer and human resources does the software require? E.g., does it take milliseconds or hours to find a requested route? Can the software run on a mobile platform with limited computer speed and memory?

---

[1] `https://en.wikipedia.org/wiki/Software_bug`

[2] `http://edison.rutgers.edu/NamesSearch/DocImage.php3?DocId=LB003487`

[3] `https://en.wikipedia.org/wiki/List_of_software_bugs`

[4] ISO/IEC 9126, International standard for the evaluation of software quality, December 19, 1991, later replaced by ISO/IEC 25010:2011

Maintainability: In case of the discovery of new bugs, is it easy to test and correct the software? Is it easy to extend the software with new functionality? E.g., is it easy to update the map with updated roadmaps and new information? Can the system be improved to work both for car drivers and bicyclists?

Portability: Is it easy to port the software to new systems such as new server architecture and screen sizes? E.g., if the routing software originally was written for IOS devices, will it be easy to port to Android systems?

The above-mentioned concepts are ordered based on the requirements of the system. Functionality and reliability are perhaps the most important concepts, since if the software does not solve the specified problem, then the software design process has failed. However, many times the problem definition will evolve along with the software development process. But as a bare minimum, the software should run without internal errors and not crash under a well-defined set of circumstances. Furthermore, it is often the case that software designed for the general public requires a lot of attention to the usability of the software, since in many cases non-experts are expected to be able to use the software with little or no prior training. On the other hand, software used internally in companies will be used by a small number of people who become experts in using the software, and it is often less important that the software is easy to understand by non-experts. An example is text processing software like Microsoft Word versus Gnu Emacs and LaTeX. Word is designed to be used by non-experts for small documents such as letters and notes and relies heavily on interfacing with the system using click-interaction. On the other hand, Emacs and LaTeX are for experts for longer and professionally typeset documents and relies heavily on keyboard shortcuts and text-codes for typesetting document entities.

The purpose of *software testing* is to find bugs. When errors are found, then we engage in *debugging*, which is the process of diagnosing and correcting bugs. Once we have a failed software test, i.e., one that does not find any bugs, then we have strengthened our belief in the software, but it is important to note that software testing and debugging rarely removes all bugs, and with each correction or change of software there is a fair risk new bugs being introduced. It is not exceptional that the testing-software is as large as the software being tested.

In this chapter, we will focus on two approaches to software testing which emphasize functionality: *white-box* and *black-box testing*. An important concept in this context is *unit testing*, where the program is considered in smaller pieces, called units, and for which accompanying programs for testing can be made which test these units automatically. Black-box testing considers the problem formulation and the program interface, and can typically be written early in the software design phase. In contrast, white-box testing considers the program text, and thus requires the program to be available. Thus, there is a tendency for black-box test programs to be more stable, while white-box testing typically is developed incrementally alongside the software development.

To illustrate software testing, we'll start with a problem:

Problem 14.1

iven any date in the Gregorian calendar, calculate the day of the week.

Facts about dates in the Gregorian calendar are:

- Combinations of dates and weekdays repeat themselves every 400 years.
- The typical length of the months January, February, . . . follow the knuckle rule, i.e., January belongs to the index knuckle, February to the space between the index and the middle finger, and August restarts or starts on the other hand. All knuckle months have 31 days, all spacing months have 30 days except February, which has 29 days on leap years and 28 days all other years.
- A leap year is a multiple of 4, except if it is also a multiple of 100 but not of 400.

Many solutions to the problem have been discovered, and here we will base our program on Gauss' method, which is based on integer division and calculates the weekday of the 1st of January of a given year. For any other date, we will count our way through the weeks from the previous 1st of January. The algorithm relies on an enumeration of weekdays starting with Sunday = 0, Monday = 1, . . . , and Saturday = 6. Our proposed solution is shown in Listing 14.1. Note that this problem has been chosen such that the solution is complicated which is a typical testing scenario, where the inner workings of the code is non-triveal.

14.1 Black-box Testing

In black-box testing, the program is considered a black box, and no knowledge is required about how a particular problem is solved. In fact, it is often useful not to have that knowledge at all. It is rarely possible to test all input to a program, so in black-box testing, the solution is tested for typical and extreme cases based on knowledge of the problem. The procedure is as follows:

1. Decide on the interface to use: It is useful to have an agreement with the software developers about what interface is to be used, e.g., in our case, the software developer has made a function `date2Day d m y` where `d`, `m`, and `y` are integers specifying the day, month, and year.
2. Make an overall description of the tests to be performed and their purpose:

Listing 14.1 date2Day.fsx:
A function that can calculate day-of-week from any date in the Gregorian calendar.

```
let januaryFirstDay (y : int) =
  let a = (y - 1) % 4
  let b = (y - 1) % 100
  let c = (y - 1) % 400
  (1 + 5 * a + 4 * b + 6 * c) % 7

let rec sum (lst : int list) j =
  if 0 <= j && j < lst.Length then
    lst[0] + sum lst[1..] (j - 1)
  else
    0

let date2Day d m y =
  let dayPrefix =
    ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri"; "Satur"]
  let feb = if (y % 4 = 0) && ((y % 100 <> 0) || (y % 400 =
   0)) then 29 else 28
  let daysInMonth = [31; feb; 31; 30; 31; 30; 31; 31; 30; 31;
   30; 31]
  let dayOne = januaryFirstDay y
  let daysSince = (sum daysInMonth (m - 2)) + d - 1
  let weekday = (dayOne + daysSince) % 7;
  dayPrefix[weekday] + "day"
```

1 a consecutive week, to ensure that all weekdays are properly returned

2 two set of consecutive days across boundaries that may cause problems: across a new year, and across a regular month boundary.

3 a set of consecutive days across February-March boundaries for a leap and non-leap year

4 four dates after February in a non-leap year, a non-multiple-of-100 leap year, a multiple-of-100-but-not-of-400 non-leap year, and a multiple-of-400 leap year.

Given no information about the program's text, there are other dates that one could consider as likely candidates for errors, but the above is judged to be a fair coverage.

3. Choose a specific set of input and expected output relations on the tabular form as shown in Table 14.1.

4. Write a program executing the tests, as shown in Listing 14.2 and 14.3. Notice how the program has been made such that it is almost a direct copy of the table produced in the previous step.

Test number	Input	Expected output
1a	1 1 2016	Friday
1b	2 1 2016	Saturday
1c	3 1 2016	Sunday
1d	4 1 2016	Monday
1e	5 1 2016	Tuesday
1f	6 1 2016	Wednesday
1g	7 1 2016	Thursday
2a	31 12 2014	Wednesday
2b	1 1 2015	Thursday
2c	30 9 2017	Saturday
2d	1 10 2017	Sunday
3a	28 2 2016	Sunday
3b	29 2 2016	Monday
3c	1 3 2016	Tuesday
3d	28 2 2017	Tuesday
3e	1 3 2017	Wednesday
4a	1 3 2015	Sunday
4b	1 3 2012	Thursday
4c	1 3 2000	Wednesday
4d	1 3 2100	Monday

Table 14.1 Black-box testing

A black-box test is a statement of what a solution should fulfill for a given problem.
★ Hence, **it is a good idea to make a black-box test early in the software design phase, in order to clarify the requirements for the code to be developed and take an outside view of the code prior to developing it.**

After the black-box testing has failed to find errors in the program, we have some confidence in the program, since, from a user's perspective, the program produces sensible output in many cases. It is, however, in no way a guarantee that the program is error free.

14.2 White-box Testing

White-box testing considers the text of a program. The degree to which the text of the program is covered in the test is called the *coverage*. Since our program is small, we have the opportunity to ensure that all functions are called at least once, which is called *function coverage*, and we will also be able to test every branching in the program, which is called *branching coverage*. If both are fulfilled, we say that we have *statement coverage*. The procedure is as follows:

1. Decide which units to test: The program shown in Listing 14.1 has 3 functions, and we will consider these each as a unit, but we might as well just have chosen

Listing 14.2 date2DayBlackTest.fsx:
The tests identified by black-box analysis. The program from Listing 14.4 has been omitted for brevity.

```
28 let testCases = [
29   ("A complete week",
30    [(1, 1, 2016, "Friday");
31     (2, 1, 2016, "Saturday");
32     (3, 1, 2016, "Sunday");
33     (4, 1, 2016, "Monday");
34     (5, 1, 2016, "Tuesday");
35     (6, 1, 2016, "Wednesday");
36     (7, 1, 2016, "Thursday");]);
37   ("Across boundaries",
38    [(31, 12, 2014, "Wednesday");
39     (1, 1, 2015, "Thursday");
40     (30, 9, 2017, "Saturday");
41     (1, 10, 2017, "Sunday")]);
42   ("Across Feburary boundary",
43    [(28, 2, 2016, "Sunday");
44     (29, 2, 2016, "Monday");
45     (1, 3, 2016, "Tuesday");
46     (28, 2, 2017, "Tuesday");
47     (1, 3, 2017, "Wednesday")]);
48   ("Leap years",
49    [(1, 3, 2015, "Sunday");
50     (1, 3, 2012, "Thursday");
51     (1, 3, 2000, "Wednesday");
52     (1, 3, 2100, "Monday")]);
53   ]
54
55 printfn "Black-box testing of date2Day.fsx"
56 for i = 0 to testCases.Length - 1 do
57   let (setName, testSet) = testCases[i]
58   printfn "  %d. %s" (i+1) setName
59   for j = 0 to testSet.Length - 1 do
60     let (d, m, y, expected) = testSet[j]
61     let day = date2Day d m y
62     printfn "    test %d - %b" (j+1) (day = expected)
```

`date2Day` as a single unit. The important part is that the union of units must cover the whole program text, and since `date2Day` calls both `januaryFirstDay` and `sum`, designing test cases for the latter two is superfluous. However, we may have to do this anyway when debugging, and we may choose at a later point to use these functions separately, and in both cases, we will be able to reuse the testing of the smaller units.

2. Identify branching points: The function `januaryFirstDay` has no branching function, `sum` has one, and depending on the input values, two paths through the code may be used, and `date2Day` has one where the number of days in February is decided. Note that in order to test this, our test-date must be March 1 or later.

Listing 14.3: Output from Listing 14.2.

```
$ dotnet fsi date2DayBlackTest.fsx
Black-box testing of date2Day.fsx
  1. A complete week
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
    test 5 - true
    test 6 - true
    test 7 - true
  2. Across boundaries
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
  3. Across Feburary boundary
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
    test 5 - true
  4. Leap years
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
```

In this example, there are only examples of `if`-branch points, but they may as well be loops and pattern matching expressions. In the Listing 14.4 it is shown that the branch points have been given a comment and a number.

3. For each unit, produce an input set that tests each branch: In our example, the branch points depend on a Boolean expression, and for good measure, we are going to test each term that can lead to branching. Using 't' and 'f' for `true` and `false`, we thus write as shown in Table 14.2. The impossible cases have been intentionally blank, e.g., it is not possible for $j < 0$ and $j > n$ for some positive value n.

4. Write a program that tests all these cases and checks the output, see Listing 14.5.

Notice that the output of the tests is organized such that they are enumerated per unit, hence we can rearrange as we like and still uniquely refer to a unit's test. Also, the output of the test program produces a list of tests that should return true or success or a similar positively loaded word, but without further or only little detail, such that we at a glance can identify any test that produced unexpected results.

Listing 14.4 date2DayAnnotated.fsx:
In white-box testing, the branch points are identified.

```
// Unit: januaryFirstDay
let januaryFirstDay (y : int) =
  let a = (y - 1) % 4
  let b = (y - 1) % 100
  let c = (y - 1) % 400
  (1 + 5 * a + 4 * b + 6 * c) % 7

// Unit: sum
let rec sum (lst : int list) j =
  (* WB: 1 *)
  if 0 <= j && j < lst.Length then
    lst[0] + sum lst[1..] (j - 1)
  else
    0

// Unit: date2Day
let date2Day d m y =
  let dayPrefix =
    ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri";
     "Satur"]
  (* WB: 1 *)
  let feb = if (y % 4 = 0) && ((y % 100 <> 0) || (y % 400
   = 0)) then 29 else 28
  let daysInMonth = [31; feb; 31; 30; 31; 30; 31; 31; 30;
   31; 30; 31]
  let dayOne = januaryFirstDay y
  let daysSince = (sum daysInMonth (m - 2)) + d - 1
  let weekday = (dayOne + daysSince) % 7;
  dayPrefix[weekday] + "day"
```

After the white-box testing has failed to find errors in the program, we have some confidence in the program, since we have run every line at least once. It is, however, in no way a guarantee that the program is error free, which is why white-box testing is often accompanied with black-box testing to be described next.

14.3 Key Concepts and Terms in This Chapter

In this chapter, we have considered two approaches to debugging code. You have seen how to:

- write supporting code for **black-box testing**, which tests for errors focussing on the intended functionality of the code and ignoring how it is constructed

Unit	Branch	Condition	Input	Expected output
`januaryFirstDay`	0	-	`2016`	`5`
`sum`	1	`0 <= j &&` ` j < lst.Length`		
	1a	`t && t`	`[1; 2; 3] 1`	`3`
	1b	`f && t`	`[1; 2; 3] -1`	`0`
	1c	`t && f`	`[1; 2; 3] 10`	`0`
	1d	`f && f`	-	-

`date2Day`	1	`(y % 4 = 0) &&`
		` ((y % 100 <> 0)`
		`  ||`
		`  (y % 400 = 0))`		

	-	`t && (t || t)`	-	-
	1a	`t && (t || f)`	`8 9 2016`	`Thursday`
	1b	`t && (f || t)`	`8 9 2000`	`Friday`
	1c	`t && (f || f)`	`8 9 2100`	`Wednesday`
	-	`f && (t || t)`	-	-
	1d	`f && (t || f)`	`8 9 2015`	`Tuesday`
	-	`f && (f || t)`	-	-
	-	`f && (f || f)`	-	-

Table 14.2 Unit test

- write a **white-box test**, which in its simplest form ensures that every line of code has been run at least once
- structure a white-box test in terms of **unit**, which is why this is sometimes called **unit testing**.

Listing 14.5 date2DayWhiteTest.fsx:
The tests identified by white-box analysis. The program from Listing 14.4 has been omitted for brevity.

```
printfn "White-box testing of date2Day.fsx"
printfn "  Unit: januaryFirstDay"
printfn "    Branch: 0 - %b" (januaryFirstDay 2016 = 5)

printfn "  Unit: sum"
printfn "    Branch: 1a - %b" (sum [1; 2; 3] 1 = 3)
printfn "    Branch: 1b - %b" (sum [1; 2; 3] -1 = 0)
printfn "    Branch: 1c - %b" (sum [1; 2; 3] 10 = 0)

printfn "  Unit: date2Day"
printfn "    Branch: 1a - %b" (date2Day 8 9 2016 =
   "Thursday")
printfn "    Branch: 1b - %b" (date2Day 8 9 2000 =
   "Friday")
printfn "    Branch: 1c - %b" (date2Day 8 9 2100 =
   "Wednesday")
printfn "    Branch: 1d - %b" (date2Day 8 9 2015 =
   "Tuesday")
```

```
$ dotnet fsi date2DayWhiteTest.fsx
White-box testing of date2Day.fsx
  Unit: januaryFirstDay
    Branch: 0 - true
  Unit: sum
    Branch: 1a - true
    Branch: 1b - true
    Branch: 1c - true
  Unit: date2Day
    Branch: 1a - true
    Branch: 1b - true
    Branch: 1c - true
    Branch: 1d - true
```

Chapter 15

Classes and Objects

Abstract In the object-oriented programming paradigm, programs are structures in classes and objects, where classes are a mixture between a type declaration and a module, and objects are values of such types. A key feature of classes is that they encapsulate both data and functions, and the paradigm has successfully facilitated the creation of very large programs. As with interface files (see Section 9.2), data and functions inside classes and objects can be hidden or be exposed to the user, where the user here is a programmer using the classes and functions. In this chapter you will learn how to

- create classes and objects which encapsulate both values and functions
- expose and hide values inside a given object
- create classes that acts as modules
- define custom operators for your objects

Object-oriented programming is a programming paradigm that focuses on objects such as a persons, places, things, events, and concepts relevant for the problem. Object-oriented programming has a rich language for describing objects and their relations, which can seem overwhelming at first, and they will be explained in detail in this and following chapters. Here is a brief overview: The main programming structures are called a *classes* and *objects*. It is useful to think of classes as user defined types and objects as values of such types. However, there is more to classes and objects than types and values. Objects may contain both data and code, and it is sometimes useful to draw the corresponding class definition as shown in Figure 15.1. In this illustration, objects of type `aClass` will each contain an int and a pair of a

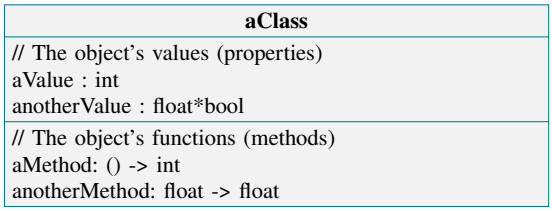


Fig. 15.1 A class is sometimes drawn as a figure.

float and a boolean, and each object has two functions associated with them. The values stored in each object may differ, but the types are fixed by the class definition. It is common to call an object's values *properties* and an object's functions *methods*. In short, properties and methods are collectively called *members*. When an object is created, memory is set aside on *The Heap* to each object's property. Creating an object is commonly called *instantiation*. The members serve as the interface to each object, and each instantiated object will have the same type of members as all objects of that class, but their content may differ.

Object-oriented programming is an extension of data types, in the sense that objects contain both data and functions in a similar manner as a module, but object-oriented programming emphasizes the semantic unity of the data and functions. Thus, objects are often *models* of real-world entities, and object-oriented programming leads to a particular style of programming analysis and design called *object-oriented analysis and design* to be discussed in Chapter 17.

15.1 Constructors and Members

A class is defined using the `type` keyword. Note that there are *always* parentheses after the class name to distinguish it from a regular type definition. The basic syntax for a class definition is as follows:

Listing 15.1: Syntax for simple class definitions.

```
1 type <classIdent> ({<arg>}) [as <selfIdent>]
2   {let <binding> | do <statement>}
3   {member <memberDef>}
```

The first line is the header of the class, where the `<classIdent>` is the name of the class, `<arg>` are its optional arguments, and `<selfIdent>` is an optional *self identifier*. The body of a class consists of the constructor and the member section. The header and the constructor section is often collectively called the *constructor*, and the body of the constructor consist of optional `let`-bindings and `do`-statements. Note that the `do`-statements in a class definition *must* use the `do`-keyword. The member section consisting of all the optional member definitions, where each definition use the `member`-keyword.

The header and constructor section is commonly called the *constructor*, and the constructor is executed at instantiation. In contrast to many other languages, the constructor is always stated as the initial code of a class definition. The values and variables in the constructor are called *fields*, while functions are just called *functions*.

Members are declared using the *`member`*-keyword, which defines values and functions that are accessible from outside the class using the "`.`"-notation. In this manner, the members define the *interface* between the internal bindings in the constructor and an application program. Member values are called *properties*, and member functions are called *methods*. Note that members are immutable. The body of a member has access to the arguments, the constructor's bindings, and to all class members, regardless of the member's lexicographical order. In contrast, members are not available in the constructor unless the self identifier has been declared in the header using the keyword *`as`*, e.g., `type classMutable(name : string) as this = ...`.

Consider the example in Figure 15.2. Here we have defined a class `car`, instanti-

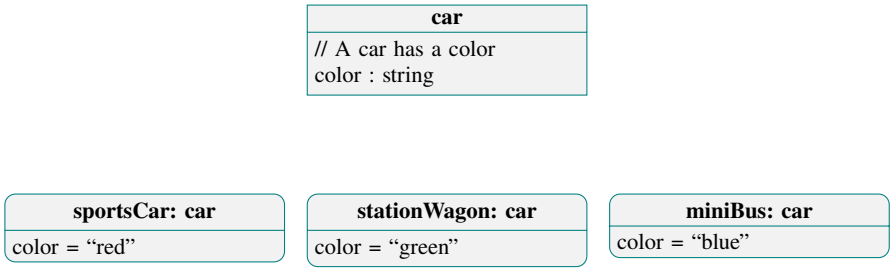


Fig. 15.2 A class `car` is instantiated trice and bound to the names `sportsCar`, `stationWagon`, and `miniBus`, and each object's properties are set to different values.

ated three objects, and bound them to the names `sportsCar`, `stationWagon`, and `miniBus`. Each object has been given different values for the `color` property. In F# this could look like the code in Listing 15.2. In the example, the class `car` is defined in lines 1–3. Its header includes one string argument, `aColor`. The body of

Listing 15.2 car.fsx:
Defining a class car, and making three instances of it. See also Figure 15.2.

```
1 type car (aColor : string) =
2   // Member section
3   member this.color = aColor
4
5 let sportsCar = car ("red")
6 let stationWagon = car ("green")
7 let miniBus = car ("blue")
8 printfn "%s %s %s" sportsCar.color stationWagon.color
     miniBus.color
```

```
$ dotnet fsi car.fsx
red green blue
```

the constructor is empty, and the member section consists of lines 2–3. The class defines one property `color : string`. Note that when referring to a member inside an object, then we must use a *self identifier*. Here we use `this` as the self identifier, and as the example shows, we need not declare it in the class' header. A self identifier refers to the memory set aside to the particular instance of an object. It is common among other programming languages to use `this` as self identifier. F# is very flexible regarding what name can be used for the self-identifier, and the member section could as well have been `self.value`, `__.value`, or anything else, and it need not be the same in every member definition. Nevertheless, **consistency in the name used as self-identifier is strongly encouraged, preferably using a name that reflects the nature of the reference, such as `this` or `me`.** The objects are instantiated in lines 5–7, and the value of their properties are accessed in line 8. In many languages, objects are instantiated using the *new* keyword, but in F# this is optional. I.e., `let sportsCar = car ("red")` is identical to `let sportsCar = new car ("red")`. Note that both the self identifier and member access uses the "`.`" notation.

★

A more advanced implementation of a `car` class might include notions of a fuel gauge, fuel economy, and the ability to update the fuel gauge as the car is driven. An example of an implementation of this is given In Listing 15.3. Here in line 1, the constructor has 2 arguments: the fuel economy parameter and the initial amount of fuel in the tank. Thus, we are able to create 2 different cars with different fuel economy, as shown in lines 10–11. The amount of fuel left en each car object is stored in the mutable field `fuelLeft`. This is an example of a state of an object: It can be accessed outside the object by the `fuel` property, and it can be updated by the `drive` method.

Field names and functions defined in the constructor do not use the self identifier and cannot be accessed outside and object using the "`.`" notation. However, they are available in both the constructor and the member section following the regular scope

Listing 15.3 class.fsx:
Extending Listing 15.2 with fields and methods.

```
type car (econ : float, fuel : float) =
  // Constructor body section
  let mutable fuelLeft = fuel // liters in the tank
  do printfn "Created a car (%.1f, %.1f)" econ fuel
  // Member section
  member this.fuel = fuelLeft
  member this.drive distance =
    fuelLeft <- fuelLeft - econ * distance / 100.0

let sport = car (8.0, 60.0)
let economy = car (5.0, 45.0)
sport.drive 100.0
economy.drive 100.0
printfn "Fuel left after 100km driving:"
printfn " sport: %.1f" sport.fuel
printfn " economy: %.1f" economy.fuel
```

```
$ dotnet fsi class.fsx
Created a car (8.0, 60.0)
Created a car (5.0, 45.0)
Fuel left after 100km driving:
 sport: 52.0
 economy: 40.0
```

rules. Fields are a common way to hide implementation details, and they are *private* to the object or class in contrast to members that are *public*.

15.2 Accessors

Methods are most often used as an interface between the fields of an object and the application program. Consider the example in Listing 15.4. In the example, the data contained in objects of type `aClass` is stored in the mutable field `v`. Since only members can be accessed from an application, it is not possible to retrieve or change the data of these object of class `aClass` directly. We could have programmed `v` as a member instead, i.e., `member this.v = 1`, however, often we are in a situation, where there is a range of possible choices of data representation, details of which we do wish to share with an application program. E.g., implementation details of arrays are not important for our ability to use them in applications. What matters is that the members that work on the array elements are well defined and efficient. Thus, the example demonstrates how we can build two simple methods `setValue` and `getValue` to set and get the data stored `v`. By making a distinction between

Listing 15.4 classAccessor.fsx:
Accessor methods interface with internal bindings.

```
type aClass () =
  let mutable v = 1
  member this.setValue (newValue : int) : unit =
    v <- newValue
  member this.getValue () : int = v

let a = aClass ()
printfn "%d" (a.getValue ())
a.setValue (2)
printfn "%d" (a.getValue ())
```

```
$ dotnet fsi classAccessor.fsx
1
2
```

the internal representation and how members give access to the data, we retain the possibility to change the internal representation without having to reprogram all the application programs. Analogously, we can change the engine in a car from one type to another without having to change the car's interaction with the driver and the road: steering wheel, pedals, wheels etc.

Such functions are called *accessors*. Internal states with setters and getters are a typical construction, since they allow for complicated computations when states are read to and written from, and gives the designer of the class the freedom to change the internal representation while keeping the interface the same. Accessors are so common that F# includes a special syntax for them: Classes can be made to act like variables using `member`...`with`...`and` keywords and the special function bindings `get()` and `set()`, as demonstrated in Listing 15.5. The expression defining `get: () -> 'a` and `set: 'a -> ()`, where `'a` is any type, can be any usual expression. The application calls the `get` and `set` as if the property were a mutable value. If `set` is omitted, then the property acts as a value rather than a variable, and values cannot be assigned to it in the application program.

Setters and getters are so common that F# has a short-hand for this using `member val value = 0 with get, set`, which creates the internal mutable value `value`, but this is discouraged in this text.

Defining an *`Item`* property with extended `get` and `set` makes objects act as indexed variables, as demonstrated in Listing 15.6. Higher dimensional indexed properties are defined by adding more indexing arguments to the definition of `get` and `set`, such as demonstrated in Listing 15.7.

Listing 15.5 classGetSet.fsx:
Members can act as variables with the built-in get and set functions.

```
type aClass () =
  let mutable v = 0
  member this.value
    with get () = v
    and set (a) = v <- a

let a = aClass ()
printfn "%d" a.value
a.value<-2
printfn "%d" a.value
```

```
$ dotnet fsi classGetSet.fsx
0
2
```

Listing 15.6 classGetSetIndexed.fsx:
Properties can act as indexed variables with the built-in get and set functions.

```
type aClass (size : int) =
  let arr = Array.create<int> size 0
  member this.Item
    with get (ind : int) = arr[ind]
    and set (ind : int) (p : int) = arr[ind] <- p

let a = aClass (3)
printfn "%A" a
printfn "%d %d %d" a[0] a[1] a[2]
a[1] <- 3
printfn "%d %d %d" a[0] a[1] a[2]
```

```
$ dotnet fsi classGetSetIndexed.fsx
FSI_0001+aClass
0 0 0
0 3 0
```

Listing 15.7 classGetSetHigherIndexed.fsx:
Getters and setters for higher dimensional index variables.

```
type aClass (rows : int, cols : int) =
  let arr = Array2D.create<int> rows cols 0
  member this.Item
    with get (i : int, j : int) = arr[i,j]
    and set (i : int, j : int) (p : int) = arr[i,j] <- p

let a = aClass (3, 3)
printfn "%A" a
printfn "%d %d %d" a[0,0] a[0,1] a[2,1]
a[0,1] <- 3
printfn "%d %d %d" a[0,0] a[0,1] a[2,1]
```

```
$ dotnet fsi classGetSetHigherIndexed.fsx
FSI_0001+aClass
0 0 0
0 3 0
```

15.3 Objects are Reference Types

Objects are reference type values, implying that copying objects copies their references, not their values, and their content is stored on *The Heap*.Consider the example in Listing 15.8. Thus, the binding to `b` in line 6 is an alias to `a`, not a copy, and changing object `a` also changes `b`! This is a common cause of error, and you should **think of objects as arrays.** For this reason, it is often seen that classes implement a copy function returning a new object with copied values, as shown in Listing 15.9. In the example, we see that since `b` now is a copy, we do not change it by changing `a`. This is called a *copy constructor*. ★

Listing 15.8 classReference.fsx:
Objects assignment can cause aliasing.

```
1 type aClass () =
2   let mutable v = 0
3   member this.value with get () = v and set (a) = v <- a
4
5 let a = aClass ()
6 let b = a
7 a.value <- 2
8 printfn "%d %d" a.value b.value
```

```
1 $ dotnet fsi classReference.fsx
2 2 2
```

Listing 15.9 classCopy.fsx:
A copy method is often needed. Compare with Listing 15.8.

```
1  type aClass () =
2    let mutable v = 0
3    member this.value with get () = v and set (a) = v <- a
4    member this.copy () =
5      let o = aClass ()
6      o.value <- v
7      o
8  let a = aClass ()
9  let b = a.copy ()
10 a.value<-2
11 printfn "%d %d" a.value b.value
```

```
1 $ dotnet fsi classCopy.fsx
2 2 0
```

15.4 Static Classes

Classes can act as modules and hold data which is identical for all objects of its type. These are defined using the *static*-keyword. And since they do not belong to a single object, but are shared between all objects, they are defined without the self-identifier and accessed using the class name, and they cannot refer to the arguments of the constructor. For example, consider a class whose objects each hold a unique identification number (id): When an object is instantiated, the object must be given the next available identification number. The next available id could be given as an argument to the constructor, however, this delegates the task of maintaining the uniqueness of ids to the application program. It is better to use a static field and delegate the administration of ids completely to the constructors, as demonstrated in Listing 15.10. Notice in line 2 that a static field `nextAvailableID` is created for the

Listing 15.10 classStatic.fsx:
Static fields and members are identical to all objects of the type.

```
1  type student (name : string) =
2    static let mutable nextAvailableID = 0 // A global id for
       all objects
3    let studentID = nextAvailableID // A per object id
4    do nextAvailableID <- nextAvailableID + 1
5    member this.id with get () = studentID
6    member this.name = name
7    static member nextID = nextAvailableID // A global member
8  let a = student ("Jon") // Students will get unique ids, when
     instantiated
9  let b = student ("Hans")
10 printfn "%s: %d,  %s: %d" a.name a.id  b.name b.id
11 printfn "Next id: %d" student.nextID // Accessing the class's
     member
```

```
1  $ dotnet fsi classStatic.fsx
2  Jon: 0,  Hans: 1
3  Next id: 2
```

value to be shared by all objects. The initialization of its value is only performed once, at the beginning of program execution. However, every time an object is instantiated, the value of `nextAvailableID` is copied to the object's field `studentID` in line 3, and `nextAvailableID` is updated. The static field can be accessed with a static accessor, as demonstrated in line 7. Notice how this definition does not include a self-identifier, and that the member is accessible from the application in line 11 using the class' name, in both cases since it is not a member of any particular object.

15.5 Recursive Members and Classes

The members of a class are inherently recursive: static and non-static methods may recurse using the self identifier and other members regardless of their lexicographical scope. This is demonstrated in Listing 15.11. For mutually recursive classes, the

Listing 15.11 classRecursion.fsx:
Members can recurse without the `rec` keyword and refer to other members regardless of their lexicographical scope.

```
type twice (v : int) =
  static member fac n = if n > 1 then n * (twice.fac (n-1))
   else 1 // No rec
  member this.copy = this.twice // No lexicographical scope
  member this.twice = 2*v

let a = twice (2)
let b = twice.fac 3
printfn "%A %A %A" a.copy a.twice b
```

```
$ dotnet fsi classRecursion.fsx
4 4 6
```

keyword *and* must be used, as shown in Listing 15.12. Here `anInt` and `aFloat`

Listing 15.12 classAssymetry.fsx:
Mutually recursive classes are defined using the `and` keyword.

```
type anInt (v : int) =
  member this.value = v
  member this.add (w : aFloat) : aFloat = aFloat ((float
   this.value) + w.value)
and aFloat (w : float) =
  member this.value = w
  member this.add (v : anInt) : aFloat = aFloat ((float
   v.value) + this.value)
let a = anInt (2)
let b = aFloat (3.2)
let c = a.add b
let d = b.add a
printfn "%A %A %A %A" a.value b.value c.value d.value
```

```
$ dotnet fsi classAssymetry.fsx
2 3.2 5.2 5.2
```

hold an integer and a floating point value respectively, and they both implement an addition of `anInt` an `aFloat` that returns and `aFloat`. Thus, they are mutually dependent and must be defined in the same `type` definition using `and`.

15.6 Function and Operator Overloading

It is often convenient to define different methods that have the same name, but with functionalities that depend on the number and type of arguments given. This is called *overloading*, and F# supports method overloading. An example is shown in Listing 15.13. In the example we define an object which can produce greetings

Listing 15.13 classOverload.fsx:
Overloading methods `set : int -> ()` and `set : int * int -> ()` is permitted, since they differ in argument number or type.

```
type Greetings () =
  let mutable greetings = "Hi"
  let mutable name = "Programmer"
  member this.str = greetings + " " + name
  member this.setName (newName : string) : unit =
    name <- newName
  member this.setName (newName : string, newGreetings :
   string) : unit =
    greetings <- newGreetings
    name <- newName
let a = Greetings ()
printfn "%s" a.str
a.setName ("F# programmer")
printfn "%s" a.str
a.setName ("Expert", "Hello")
printfn "%s" a.str
```

```
$ dotnet fsi classOverload.fsx
Hi Programmer
Hi F# programmer
Hello Expert
```

strings of the form `<greeting> <name>`, using the `str` member. It has a default greeting "Hi" and name "Programmer", but the name can be changed by calling the `setName` accessor with one argument, and both greeting and name can be changed by calling the overloaded `setName` with two arguments. Overloading in class definition is allowed as long as the arguments differ in number or type.

In Listing 15.12, the notation for addition is less than elegant. For such situations, F# supports *operator overloading*. All usual operators may be overloaded, and the compiler uses type inference to decide which function is to be called. All operators have a functional equivalence, and to overload the binary "+" and unary "-" operators, we overload their functional equivalence `(+)` and `(~-)` as static members. This is demonstrated in Listing 15.14. Thus, writing `v + w` is equivalent to writing `anInt.(+) (v, w)`. Presently, the former is to be preferred, but at times, e.g., when using functions as arguments, it is useful to be able to refer to an operator

Listing 15.14 classOverloadOperator.fsx:
Operators can be overloaded using their functional equivalents.

```
type anInt (v : int) =
  member this.value = v
  static member (+) (v : anInt, w : anInt) = anInt (v.value +
   w.value)
  static member (~-) (v : anInt) = anInt (-v.value)
and aFloat (w : float) =
  member this.value = w
  static member (+) (v : aFloat, w : aFloat) = aFloat
   (v.value + w.value)
  static member (+) (v : anInt, w : aFloat) =
    aFloat ((float v.value) + w.value)
  static member (+) (w : aFloat, v : anInt) = v + w // reuse
   def. above
  static member (~-) (v : aFloat) = aFloat (-v.value)

let a = anInt (2)
let b = anInt (3)
let c = aFloat (3.2)
let d = a + b // anInt + anInt
let e = c + a // aFloat + anInt
let f = a + c // anInt + aFloat
let g = -a // unitary minus anInt
let h = a + -b // anInt + unitary minus anInt
printf "a=%A, b=%A, c=%A, d=%A" a.value b.value c.value
   d.value
printf ", e=%A, f=%A, g=%A, h=%A" e.value f.value g.value
   h.value
```

```
$ dotnet fsi classOverloadOperator.fsx
a=2, b=3, c=3.2, d=5, e=5.2, f=5.2, g=-2, h=-1
```

by its function-equivalent. Note that the functional equivalence of the multiplication operator `(*)` shares a prefix with the begin block comment lexeme "`(*`", which is why the multiplication function is written as `( * )`. Note also that unitary operators have a special notation using the "~"-lexeme, as illustrated in the above example for unitary minus. With the unitary minus, we are able to subtract objects of `anInt` by first negating the right-hand operand and then adding the result to the left-hand operand. In contrast, the binary minus would have been defined as `static member (-) (v : anInt, w : aFloat) = anInt ((float v.value) - w.value)`.

In Listing 15.14, notice how the second `(+)` operator overloads the first by calling the first with the proper order of arguments. This is a general principle: **avoid duplication of code, reuse of existing code is almost always preferred.** Here it is to be preferred for two reasons. Firstly, if we discover a mistake in the multiplication code, then we need only correct it once, which implies that both multiplication methods are corrected once and reduces the chance of introducing new mistakes by attempting to ★

correct old ones. Secondly, if we later decide to change the internal representation, then we only need to update one version of the multiplication function, hence we reduce programming time and risk of errors as well.

Beware that operator overloading outside class definitions overwrites *all* definitions of the operator. E.g., overloading `(+) (v, w)` outside a class will influence integer,
★ real, string, etc. Thus, **operator overloading should only be done inside class definitions.**

15.7 Additional Constructors

Like methods, constructors can also be overloaded by using the *new* keyword. E.g., the example in Listing 15.13 may be modified, such that the name and possibly greeting is set at object instantiation rather than by using the accessor. This is illustrated in Listing 15.15. The top constructor that does not use the `new`-keyword is called the *pri-*

Listing 15.15 classExtraConstructor.fsx:
Extra constructors can be added, using `new`.

```
type classExtraConstructor (name : string, greetings :
    string) =
  static let defaultGreetings = "Hello"
 // Additional constructors are defined by new ()
  new (name : string) =
    classExtraConstructor (name, defaultGreetings)
  member this.name = name
  member this.str = greetings + " " + name

let s = classExtraConstructor ("F#") // Calling additional
    constructor
let t = classExtraConstructor ("F#", "Hi") // Calling primary
    constructor
printfn "%A, %A" s.str t.str
```

```
$ dotnet fsi classExtraConstructor.fsx
"Hello F#", "Hi F#"
```

mary constructor. The body of the additional constructor must call the primary constructor, and the body cannot extend the primary constructor's fields and functions. It
★ is useful to **think of the primary constructor as a superset of arguments and the additional ones as subsets or specializations.** As regular scope rules dictate, the additional constructor has access to the primary constructor's bindings. However, in order to access the object's members, the self identifier has to be explicitly declared, using the `as`-keyword in the header. E.g., writing `new(x : float, y : float)`

`as alsoThis = ...`. However beware. Even though the body of the additional constructor now may access the property `alsoThis.x`, this value has first been created once the primary constructor has been called. E.g., calling the primary constructor in the additional constructor as `new(x : float, y : float) as alsoThis = classExtraConstructor(fst alsoThis.x, y, defaultSeparator)` will result in an exception at runtime. Code may be executed in additional constructors: Before the call to the primary constructor, `let` and `do` statements are allowed. If code is to be executed after the primary constructor has been called, then it must be preceded by the `then` keyword, as shown in Listing 15.16. The `do`-keyword is often

Listing 15.16 classDoThen.fsx:
The optional `do`- and `then`-keywords allow for computations before and after the primary constructor is called.

```
type classDoThen (aValue : float) =
  // "do" is mandatory to execute code in the primary
   constructor
  do printfn "  Primary constructor called"
  // Some calculations
  do printfn "  Primary done" (* *)
  new () =
    // "do" is optional in additional constructors
    printfn "  Additional constructor called"
    classDoThen (0.0)
    // Use "then" to execute code after construction
    then
      printfn "  Additional done"
  member this.value = aValue

printfn "Calling additional constructor"
let v = classDoThen ()
printfn "Calling primary constructor"
let w = classDoThen (1.0)
```

```
$ dotnet fsi classDoThen.fsx
Calling additional constructor
  Additional constructor called
  Primary constructor called
  Primary done
  Additional done
Calling primary constructor
  Primary constructor called
  Primary done
```

understood to be implied by F#, e.g., in front of all `printf`-statements, but in the above examples they are required where used. This may change in future releases of F#. F# allows for many additional constructors, but they must be distinguishable by type.

15.8 Programming Intermezzo: Two Dimensional Vectors

Consider the following problem.

Problem 15.1

Euclidean vector is a geometric object that has a direction, a length, and two operations: vector addition and scalar multiplication, see Figure 15.3. Define a class for a vector in two dimensions.

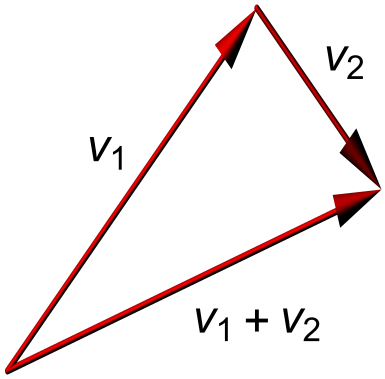


Fig. 15.3 Illustration of vector addition in two dimensions.

An essential part in designing a solution for the above problem is to decide which representation to use internally for vectors. The Cartesian representation of a vector is as a tuple of real values (x, y), where x and y are real values, and where we can imagine that the tail of the vector is in the origin, and its tip is at the coordinate (x, y). For vectors on Cartesian form,

$$\mathbf{v} = (x, y), \tag{15.1}$$

the basic operations are defined as

$$\mathbf{v}_1 + \mathbf{v}_2 = (x_1 + x_2, y_1 + y_2), \tag{15.2}$$

$$a\mathbf{v} = (ax, ax), \tag{15.3}$$

$$\text{dir}(\mathbf{v}) = \tan \frac{y}{x},\ x \neq 0, \tag{15.4}$$

$$\text{len}(\mathbf{v}) = \sqrt{x^2 + y^2}, \tag{15.5}$$

where x_i and y_i are the elements of vector $\mathbf{v}_i$, a is a scalar, and dir and len are the direction and length functions, respectively. The polar representation of vectors is also a tuple of real values (θ, l), where θ is the vector's angle from the x-axis and l is the vector's length. This representation is closely tied to the definition of a vector, and has the constraint that $0 \leq \theta < 2\pi$ and $0 \leq l$. This representation reminds us

that vectors do not have a position. For vectors on polar form,

$$\mathbf{v} = (\theta, l), \tag{15.6}$$

their basic operations are defined as

$$x(\theta, l) = l\cos(\theta), \tag{15.7}$$
$$y(\theta, l) = l\sin(\theta), \tag{15.8}$$
$$\mathbf{v}_1 + \mathbf{v}_2 = (x(\theta_1, l_1) + x(\theta_2, l_2), y(\theta_1, l_1) + y(\theta_2, l_2)) \tag{15.9}$$
$$a\mathbf{v} = (\theta, al), \tag{15.10}$$

where θ_i and l_i are the elements of vector $\mathbf{v}_i$, a is a scalar, and x and y are the Cartesian coordinate functions.

So far in our analysis, we have realized that:

- both the Cartesian and polar representations use a pair of reals to represent the vector,
- both require functions to calculate the elements of the other representation,
- the polar representation is invalid for negative lengths, and
- the addition operator under the polar representation is also more complicated and essentially requires access to the Cartesian representation.

The first step in shaping our solution is to decide on file structure: For conceptual separation, we choose to use a library and an application file. F# wants files to define namespaces or modules, so we choose the library to be a `Geometry` module, which implements the vector class to be called `vector`. Furthermore, when creating vector objects we would like to give the application program the ability to choose either Cartesian or polar form. This is can be done using *discriminated unions*. Discriminated unions allow us to tag values of possibly identical form, but they also lead to longer programs. Thus, we will also provide an additional constructor on implicit Cartesian form, since this is the most common representation of vectors.

A key point when defining libraries is to consider their interface with the application program. Hence, our second step is to write an application using the yet to be written library in order to get a feel for how such an interface could be. This is demonstrated in the application program Listing 15.17. The application of the vector class seems natural, makes use of the optional discriminated unions, uses the infix operators "+" and "*" in a manner close to standard arithmetic, and interacts smoothly with the `printf` family. Thus, we have further sketched requirements to the library with the emphasis on application.

Listing 15.17 vectorApp.fsx:
An application using the library in Listing 15.18.

```
open Geometry
let v = vector(Cartesian (1.0,2.0))
let w = vector(Polar (3.2,1.8))
let p = vector()
let q = vector(1.2, -0.9)
let a = 1.5
printfn "%A * %A = %A" a v (a * v)
printfn "%A + %A = %A" v w (v + w)
printfn "vector() = %A" p
printfn "vector(1.2, -0.9) = %A" q
printfn "v.dir = %A" v.dir
printfn "v.len = %A" v.len
```

After a couple of trials, our library implementation has ended up as shown in Listing 15.18. Realizations achieved during writing this code are: Firstly, in order to implement a vector class using discriminated unions, we had to introduce a constructor with helper variables `_x`, `_y`, etc. The consequence is that the Cartesian and polar representation is evaluated once and only once every time an object is created. Unfortunately, discriminated unions do not implement guards on subsets, so we still have to cast an exception when the application attempts to create an object with a negative length. Secondly, for the `ToString` override we have implemented static members for typesetting vectors, since it seems more appropriate that all vectors should be typeset identically. Changing typesetting thus respects dynamic scope.

The output of our combined library and application is shown in Listing 15.19. The output is as expected, and for the vector class, our solution seems to be a good compromise between versatility and syntactical bloating.

15.9 Key Concepts and Terms in This Chapter

In this chapter, we have introduced the concept of object-oriented programming and taken a close look at how to define classes and objects and some of the ways to use them in F#. Particularly you have learned:

- how to create a **class** and make values of the class which are called **objects**. An object is often called an **instance** of a class, and the process of creating objects is called **instantiation**. The **constructor** is a piece of code inside a class, which is run when an object is instantiated.

Listing 15.18 vector.fs:
A library serving the application in Listing 15.19.

```
module Geometry
type Coordinate =
  Cartesian of float * float // (x, y)
  | Polar of float * float // (dir, len)
type vector(c : Coordinate) =
  let (_x, _y, _dir, _len) =
    match c with
      Cartesian (x, y) ->
        (x, y, atan2 y x, sqrt (x * x + y * y))
      | Polar (dir, len) when len >= 0.0 ->
        (len * cos dir, len * sin dir, dir, len)
      | Polar (dir, _) ->
        failwith "Negative length in polar representation."
  new(x : float, y : float) =
    vector(Cartesian (x, y))
  new() =
    vector(Cartesian (0.0, 0.0))
  member this.x = _x
  member this.y = _y
  member this.len = _len
  member this.dir = _dir
  static member val left = "(" with get, set
  static member val right = ")" with get, set
  static member val sep = ", " with get, set
  static member ( * ) (a : float, v : vector) : vector =
    vector(Polar (v.dir, a * v.len))
  static member ( * ) (v : vector, a : float) : vector =
    a * v
  static member (+) (v : vector, w : vector) : vector =
    vector(Cartesian (v.x + w.x, v.y + w.y))
  override this.ToString() =
    sprintf "%s%A%s%A%s" vector.left this.x vector.sep this.y
   vector.right
```

Listing 15.19: Compiling and running the code from Listing 15.18 and 15.17.

```
$ dotnet fsi vector.fs vectorApp.fsx
1.5 * (1.0, 2.0) = (1.5, 3.0)
(1.0, 2.0) + (-1.796930596, -0.1050734582) = (-0.7969305964,
   1.894926542)
vector() = (0.0, 0.0)
vector(1.2, -0.9) = (1.2, -0.9)
v.dir = 1.107148718
v.len = 2.236067977
```

- that the values and function **fields** and **functions**, and elements which can be accessed from the outside of an object are called **members** and are accessed using the "`.`"-notation.

- how to create a class with **static** elements which are global w.r.t. the class much like a module's elements.
- how to write the **accessors set** and **get** and how to write classes which makes the "`[]`"-indexing available for objects.
- that objects are reference types and therefore risk **aliasing**, and therefore, must often be **cloned** if an independent copy is needed.
- how to **overload** F#'s operators to generate new syntax similar to the arithmetic operators of floats.
- how to overload the instantiation of objects with additional constructors, which can be used to supply default values to the instantiation process.

Chapter 16

Derived Classes

Abstract Sometimes, classes can be grouped into a hierarchy such as foxes and fish both being animals. In this chapter, you will learn how object-oriented programming uses inheritance to benefit from such a hierarchy. For example, super-classes such as animals may contain elements, which are shared by the sub-classes foxes and fish, e.g., they both have a weight, and the sub-classes need only implement the differences, e.g., foxes live on land and fish in water. In this chapter you will learn how to

- construct a class hierarchy
- utilize the common super-class to have simple polymorphy, e.g., make lists that contains both foxes and fish
- create abstract classes similar to interface files, for a user (programmer) to later produce an implementation
- make and use class-interfaces to make your objects useful in generic functions such as sorting

16.1 Inheritance

Sometimes it is useful to derive new classes from old ones in order to reuse code or to emphasize a program structure. For example, consider the concepts of a *car* and *bicycle*. They are both *vehicles* that can move forward and turn, but a car can move in reverse, has 4 wheels, and uses gasoline or electricity, while a bicycle has 2 wheels and needs to be pedaled. Structurally, we can say that "a car is a vehicle" and "a bicycle is a vehicle". Such a relation is sometimes drawn as a tree as shown in Figure 16.1 and is called an *is-a relation*. Is-a relations can be implemented using

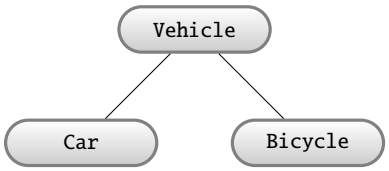


Fig. 16.1 Both a car and a bicycle is a (type of) vehicle.

class *inheritance*, where vehicle is called the *base class*, and car and bicycle are each a *derived class*. The advantage is that a derived class can inherent the members of the base class, *override*, and possibly add new members. Another advantage is that objects from derived classes can be made to look like as if they were objects of the base class while still containing all their data. Such mascarading is useful when, for example, listing cars and bicycles in the same list.

In F#, inheritance is indicated using the `inherit` keyword in the class definition. An extensions of the syntax in Listing 16.1 is:

Listing 16.1: A class definition with inheritance.

```
type <classIdent> ({<arg>}) [as <selfIdent>]
  [inherit <baseClassIdent>({<arg>})]
  {[let <binding>] | [do <statement>]}
  {(member | abstract member | default | override)
   <memberDef>}
```

New syntactical elements are: the `inherit` keyword, which indicates that this is a derived class and where `<baseClassIdent>` is the name of the base class. Further, members may be regular members using the `member` keyword as discussed in the previous chapter, and members can also be other types, as indicated by the keywords: `abstract member`, `default`, and `override`.

An example of defining base and derived classes for vehicles is shown In Listing 16.2. In the example, a simple base class `vehicle` is defined to include `wheels` as its single member. The derived classes inherit all the members of the base class, but do not have access to any non-members of the base constructor. I.e., `car` and `bicycle` automatically have the `wheels` property. Both derived classes additional members `maxPassengers` and `mustUseHelmet`, respectively.

Listing 16.2 vehicle.fsx:
New classes can be derived from old ones.

```
/// All vehicles have wheels
type vehicle (nWheels : int ) =
  member this.wheels = nWheels

/// A car is a vehicle with 4 wheels
type car (nPassengers : int) =
  inherit vehicle (4)
  member this.maxPassengers = nPassengers

/// A bicycle is a vehicle with 2 wheels
type bicycle () =
  inherit vehicle (2)
  member this.mustUseHelmet = true

let aVehicle = vehicle (1)
let aCar = car (4)
let aBike = bicycle ()
printfn "aVehicle has %d wheel(s)" aVehicle.wheels
printfn "aCar has %d wheel(s) with room for %d passenger(s)"
   aCar.wheels aCar.maxPassengers
printfn "aBike has %d wheel(s). Is helmet required? %b"
   aBike.wheels aBike.mustUseHelmet
```

```
$ dotnet fsi vehicle.fsx
aVehicle has 1 wheel(s)
aCar has 4 wheel(s) with room for 4 passenger(s)
aBike has 2 wheel(s). Is helmet required? true
```

Derived classes can replace base class members by defining new members *overshadow* the base members. The base members are still available through the `base`-keyword. Consider the example in the Listing 16.3. In this case, we have defined three greetings: `greeting`, `hello`, and `howdy`. The two later inherit `member this.str = "hi"` from `greeting`, but since they both also define a member property `str`, these overshadow the one from `greeting`. In `hello` and `howdy` the base value of `str` is available as `base.str`.

Even though derived classes are different from their base, the derived class includes the base class, which can be recalled using *upcasting* by the upcast operator *":>"*. At compile-time, this operator removes the additions and overshadowing of the derived class, as illustrated in Listing 16.4. Here `howdy` is derived from `hello`, overshadows `str`, and adds property `altStr`. By upcasting object `b`, we create object `c` as a copy of `b` with all its fields, functions, and members, as if it had been of type `hello`. I.e., `c` contains the base class version of `str` and does not have property `altStr`. Objects `a` and `c` are now of same type and can be put into, e.g., an array as `let arr = [|a, c|]`. Previously upcasted objects can also be downcasted again using the *downcast*

Listing 16.3 memberOvershadowingVar.fsx:
Inherited members can be overshadowed, but we can still access the base member. Compare with Listing 16.7.

```
/// hi is a greeting
type greeting () =
  member this.str = "hi"
/// hello is a greeting
type hello () =
  inherit greeting ()
  member this.str = "hello"
/// howdy is a greeting
type howdy () =
  inherit greeting ()
  member this.str = base.str + " there"

let a = greeting ()
let b = hello ()
let c = howdy ()
printfn "%s, %s, %s" a.str b.str c.str
```

```
$ dotnet fsi memberOvershadowingVar.fsx
hi, hello, hi there
```

★ operator *:?>*, but the validity of the operation is checked at runtime. Thus, **avoid downcasting when possible.**

Listing 16.4 upCasting.fsx:
Objects can be upcasted resulting in an object to appear to be of the base type. Implementations from the derived class are ignored.

```
/// hello holds property str
type hello () =
  member this.str = "hello"
/// howdy is a hello class and has property altStr
type howdy () =
  inherit hello ()
  member this.str = "howdy"
  member this.altStr = "hi"

let a = hello ()
let b = howdy ()
let c = b :> hello // a howdy object as if it were a hello
   object
printfn "%s %s %s %s" a.str b.str b.altStr c.str
```

```
$ dotnet fsi upCasting.fsx
hello howdy hi hello
```

16.2 Interfacing with the `printf` Family

In previous examples, we accessed the property in order to print the contents of objects. Luckily, a more elegant solution is available. Objects can be printed directly, but the result is most often not very useful, as can be seen in Listing 16.5. All classes

Listing 16.5 classPrintf.fsx:
Printing classes yields low-level information about the class.

```
type vectorDefaultToString (x : float, y : float) =
  member this.x = (x,y)

let v = vectorDefaultToString (1.0, 2.0)
printfn "%A" v // Printing objects gives low-level
   information
```

```
$ dotnet fsi classPrintf.fsx
FSI_0001+vectorDefaultToString
```

implicitely inherit from a class with the peculiar name, *`System.Object`*, and as a consequence, all classes have a number of already defined members. One example is the `ToString() : () -> string` function, which is useful in conjunction with, e.g., `printf`. When an object is given as argument to a `printf` function with the `%A` or `%O` placeholders in the formatting string, `printf` calls the object's `ToString()` function. The default implementation returns low-level information about the object, as can be seen above, but we may *override* this member using the *`override`*-keyword, as demonstrated in Listing 16.6. Note, despite that `ToString()` returns a string, the `%s` placerholder only accepts values of the basic `string` type. We see

Listing 16.6 classToString.fsx:
Overriding `ToString()` function for better interaction with members of the `printf` family of procedures. Compare with Listing 16.5.

```
type vectorWToString (x : float, y : float) =
  member this.x = (x,y)
  // Custom printing of objects by overriding this.ToString()
  override this.ToString() =
    sprintf "(%A, %A)" (fst this.x) (snd this.x)

let v = vectorWToString(1.0, 2.0)
printfn "%A" v // No change in application but result is
   better
```

```
$ dotnet fsi classToString.fsx
(1.0, 2.0)
```

that as a consequence, the `printf` statement is much simpler. However beware, an

application program may require other formatting choices than selected at the time of designing the class, e.g., in our example, the application program may prefer square
★ brackets as delimiters for vector tuples. So in general **when designing an override to `ToString()`, choose simple, generic formatting for the widest possible use.**

The most generic formatting is not always obvious, and in the vector case some candidates for the formatting string of `ToString()` are "`%A %A`", "`%A, %A`", "`(%A, %A)`", and "`[%A, %A]`". Considering each carefully, it seems that arguments can be made against all them. A common choice is to let the formatting be controlled by static members that can be changed by the application program through accessors.

16.3 Abstract Classes

In the previous sections, we have discussed inheritance as a method to modify and extend any class. I.e., the definition of the base classes were independent of the definitions of inherited classes. In that sense, the base classes were oblivious to any future derivation of them. Sometimes it is useful to define base classes which are not independent of derived classes and which impose design constraints of derived classes. Two such dependencies in F# are abstract classes and interfaces.

An *abstract class* contains members defined using the *`abstract member`* and optionally the *`default`* keywords. An `abstract member` in the base class is a type definition, and derived classes must provide an implementation using the *`override`* keyword. Optionally, the base class may provide a default implementation using the `default` keyword, in which case overriding is not required in derived classes. Objects of classes containing abstract members without default implementations cannot be instantiated, but derived classes that provide the missing implementations can. Note that abstract classes must be given the *`[<AbstractClass>]`* attribute. Note also that in contrast to overshadowing, upcasting keeps the implementations of the derived classes. Examples of this are shown in Listing 16.7. In the example, we define a base class and two derived classes. Note how the abstract member is defined in the base class using the "`:`"-operator as a type declaration rather than a name binding. Note also that since the base class does not provide a default implementation, the derived classes supply an implementation using the `override`-keyword. In the example, objects of `baseClass` cannot be created, since such objects would have no implementation for `this.hello`. Finally, the two different derived and upcasted objects can be put in the same array, and when calling their implementation of `this.hello`, we still get the derived implementations, which is in contrast to overshadowing.

Abstract classes may also specify a default implementation, such that derived classes have the option of implementing an overriding member, but are not forced to. In spite

Listing 16.7 abstractClass.fsx:
In contrast to regular objects, upcasted derived objects use the derived implementation of abstract methods. Compare with Listing 16.3.

```
/// An abstract class for general greeting classes with property str
[<AbstractClass>]
type greeting () =
  abstract member str : string
/// hello is a greeting
type hello () =
  inherit greeting ()
  override this.str = "hello"
/// howdy is a greeting
type howdy () =
  inherit greeting ()
  override this.str = "howdy"

let a = hello ()
let b = howdy ()
let c = [| a :> greeting; b :> greeting |] // arrays of greetings
Array.iter (fun (elm : greeting) -> printfn "%s" elm.str) c
```

```
$ dotnet fsi abstractClass.fsx
hello
howdy
```

of implementations being available in the abstract class, the abstract class still cannot be used to instantiate objects. The example in Listing 16.8 shows an extension of Listing 16.7 with a default implementation. In the example, the program in Listing 16.7 has been modified such that `greeting` is given a default implementation for `str`, in which case `hello` does not need to supply one. However, in order for `howdy` to provide a different greeting, it still needs to provide an override member.

Note that even if all abstract members in an abstract class have defaults, objects of its type can still not be created, but must be derived as, e.g., shown with `hello` above.

As a side note, every class implicitly derives from a base class *`System.Object`*, which is an abstract class defining among other members, the `ToString` method with default implementation.

Listing 16.8 abstractDefaultClass.fsx:
Default implementations in abstract classes make implementations in derived classes optional. Compare with Listing 16.7.

```
/// An abstract class for general greeting classes with property str
[<AbstractClass>]
type greeting () =
  abstract member str : string
  default this.str = "hello" // Provide default implementation
/// hello is a greeting
type hello () =
  inherit greeting ()
/// howdy is a greeting
type howdy () =
  inherit greeting ()
  override this.str = "howdy"

let a = hello ()
let b = howdy ()
let c = [| a :> greeting; b :> greeting |] // arrays of greetings
Array.iter (fun (elm : greeting) -> printfn "%s" elm.str) c
```

```
$ dotnet fsi abstractDefaultClass.fsx
hello
howdy
```

16.4 Interfaces

Inheritance of an abstract base class allows an application to rely on the definition of the base, regardless of any future derived classes. This gives great flexibility, but at times even less knowledge is needed about objects in order to write useful applications. This is what *interfaces* offer. An interface specifies which members must exist, but nothing more. Interfaces are defined as an abstract class *without arguments* and *only with abstract members*. Classes implementing interfaces must specify implementations for the abstract members using the *`interface with`* keywords. Objects of classes implementing interfaces can be upcasted as if they had an abstract base class of the interface's name. Consider the example in Listing 16.9. Here, two distinctly different classes are defined: `house` and `person`. These are not related by inheritance, since no sensible common structure seems available. However, they share structures in the sense that they both have an integer property and a `float -> float` method. For each of the derived classes, these members have different meanings. Still, some treatment of these members by an application will only rely on their type and not their meaning. E.g., in Listing 16.9, the `printfn` function only needs to know the member's type, not its meaning. As a consequence,

Listing 16.9 classInterface.fsx:
Interfaces specify which members classes contain, and with upcasting gives more flexibility than abstract classes.

```
/// An interface for classes that have method fct and member value
type IValue =
  abstract member fct : float -> float
  abstract member value : int
/// A house implements the IValue interface
type house (floors: int, baseArea: float) =
  interface IValue with
    // calculate total price based on per area average
    member this.fct (pricePerArea : float) =
      pricePerArea * (float floors) * baseArea
    // return number of floors
    member this.value = floors
/// A person implements the IValue interface
type person(name : string, height: float, age : int) =
  interface IValue with
    // calculate body mass index (kg/(m*m)) using hypothetic mass
    member this.fct (mass : float) = mass / (height * height)
    // return the length of name
    member this.value = name.Length
  member this.data = (name, height, age)

let a = house(2, 70.0) // a two storage house with 70 m*m base area
let b = person("Donald", 1.8, 50) // a 50 year old person 1.8 m high
let lst = [a :> IValue; b :> IValue]
let printInterfacePart (o : IValue) =
  printfn "value = %d, fct(80.0) = %g" o.value (o.fct 80.0)
List.iter printInterfacePart lst
```

```
$ dotnet fsi classInterface.fsx
value = 2, fct(80.0) = 11200
value = 6, fct(80.0) = 24.6914
```

the application can upcast them both to the implicit abstract base class `IValue`, put them in an array, and apply a function using the member definition of `IValue` with the higher-order `List.iter` function. Another example could be a higher-order function calculating average values: For average values of the number of floors and average value of the length of people's names, the higher-order function would only need to know that both of these classes implement the `IValue` interfaces in order to calculate the average of list of either objects' types.

As a final note, inheritance ties classes together in a class hierarchy. Abstract members enforce inheritance and impose constraints on the derived classes. Like abstract

classes, interfaces impose constraints on derived classes, but without requiring a hierarchical structure.

16.5 Programming Intermezzo: Chess

To demonstrate the use of hierarchies, consider the following problem.

Problem 16.1

he game of chess is a turn-based game for two which consists of a board of 8×8 squares, and a set of 16 black and 16 white pieces. A piece can be either a king, queen, rook, bishop, knight or pawn, and each piece has a specific movement pattern on the board. Pieces are added to, moved on, and removed from the board during the game, and there can be at most one piece per square. A piece strikes another piece of opposing color by moving to its square and the piece of opposing color is removed from the game. The game starts with the configuration shown in Figure 16.2.

Make a program that allows two humans to play simple chess using only kings and rooks. The king must be able to move to all neighboring squares not occupied by a piece of the same color and cannot move onto a square where it can be struck in the next turn. The rook must be able to move in horizontal and vertical lines until a piece of the same color or up to and including a piece of opposing color.

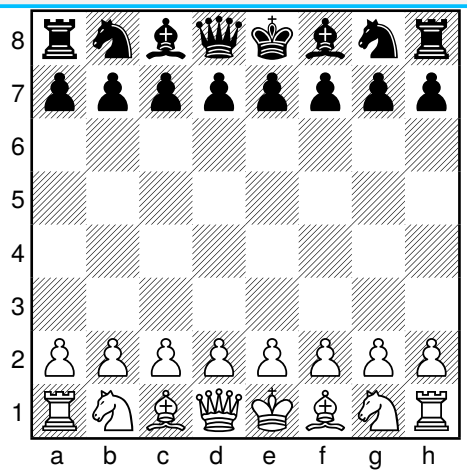


Fig. 16.2 Starting position for the game of chess.

Since we expect that the solution to the above problem is going to be a relatively long program, we have decided to split the code into a library and an application program. Before writing a library, it is often useful to start thinking about how the library should be used. Thus we start by sketching the application program, and in the process consider options for the main methods and properties to be used.

We also foresee future extensions to include more pieces, but also that these pieces will obey the same game mechanics that we design for the present problem. Thus,

we will put the main part of the library in a file defining the module called `Chess` and the derived pieces in another file defining the module `Pieces`.

Every game needs a board, and we will define a class `Board`. A board is like an array, so it seems useful to be able to move pieces by index notation. Thus, the board must have a two-dimensional `Item` property. We also decide that each position will hold an option type, such that when a square is empty it holds `None`, and otherwise it holds piece `p` as `Some p`. Although chess notation would be neat, for ease of programming we will let index (0,0) correspond to position a1 in chess notation, etc. The most common operation will probably be to move pieces around, so we will give the board a `move` method. We will most likely also like to print the board with pieces in their right locations. For simplicity, we choose to override the `ToString` method in `Board`, and that this method also prints information about each individual piece, such as where it is, where it can move to, and which pieces it can either protect or hit. The pieces that a piece can protect or hit we will call the piece's neighbor pieces.

A piece can be one of several types, so this gives a natural hierarchical structure which is well suited for inheritance. Each piece must be given a color, which may conveniently be given as argument at instantiation. Thus, we have decided to make a base class called `chessPiece` with argument `Color`, and derived classes `king` and `rook`. The color may conveniently be define as a discriminated union type of either `White` or `Black`. Each piece will also override the `ToString` method for ease of printing. The override will be used in conjunction with the board's override, so it should only give information about the piece's type and color. For compact printing, we will use a single letter for the type of piece, upper case if white, and lower case if black. We expect the pieces also to need to know something about the their relation to board, so we will make a `position` property which holds the coordinates of the piece, and we will make a `availableMoves` method that lists the possible moves a piece can make. Thus, we produce the application in Listing 16.10, and an illustration of what the program should do is shown in Figure 16.3. At this point, we are fairly happy with the way the application is written. The double bookkeeping of pieces in an array and on the board seems a bit excessive, but for testing it seems useful to be able to easily access all pieces, both those in play and struck. Although the `position` property of a `chessPiece` could be replaced by a function searching for a specific piece on the board, we have a hunch that we will need to retrieve a piece's position often, and that this double bookkeeping will most likely save execution time later.

Continuing our outer to inner approach, as a second step, we consider the specific pieces: They will inherit a base piece and implement the details that are special for that piece. Each piece is signified by its color and its type, and each type has a specific motion pattern. Since we have already decided to use discriminated unions for the color, it seems natural to let the color be part of the constructor of the base class. As in the example application in Listing 16.10, pieces are upcasted to `chessPiece`, thus, the base class must know how to print the piece type. For this, we will define an

Listing 16.10 chessApp.fsx:
A chess application.

```
open Chess
open Pieces
/// Print various information about a piece
let printPiece (board : Board) (p : chessPiece) : unit =
  printfn "%A: %A %A" p p.position (p.availableMoves board)

// Create a game
let board = Chess.Board () // Create a board
// Pieces are kept in an array for easy testing
let pieces = [|
  king (White) :> chessPiece;
  rook (White) :> chessPiece;
  king (Black) :> chessPiece |]
// Place pieces on the board
board[0,0] <- Some pieces[0]
board[1,1] <- Some pieces[1]
board[4,1] <- Some pieces[2]
printfn "%A" board
Array.iter (printPiece board) pieces

// Make moves
board.move (1,1) (3,1) // Moves a piece from (1,1) to (3,1)
printfn "%A" board
Array.iter (printPiece board) pieces
```

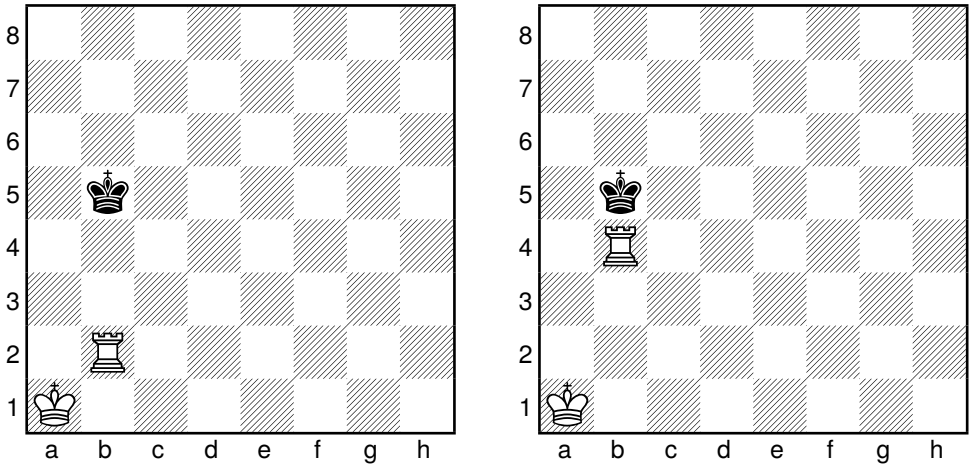


Fig. 16.3 Starting at the left and moving white rook to b4.

abstract property, such that everything needed for overriding `ToString` is available to the base class, but also such that the name of the type of the piece is set in the derived class.

For a piece on the board, its available moves depend on its type and the other pieces. The application program will need to make a decision on whether to move the piece depending on which vacant squares it can move to, and its relation to its neighbors, i.e., is the piece protecting one of its own color, or does it have the opportunity to hit an opponent's piece. Thus, given the board with all the pieces, it seems useful that `availableMoves` returns two lists: a list of vacant squares and a list of neighboring pieces of either color. Each piece has a certain movement pattern which we will specify regardless of the piece' position on the board and relation to other pieces. Thus, this will be an abstract member called `candidateRelativeMoves` implemented in the derived pieces. These candidate relative moves are then to be sifted for legal moves, and the process will be the same for all pieces. Thus, sifting can be implemented in the base class as the `availableMoves`.

Many pieces move in runs, e.g., the rook can move horizontally and vertically until there is another piece. Vacant squares behind the blocking piece are unavailable. For a rook, we must analyze four runs: northward, eastward, southward, and westward. For each run, we must consult the board to see how many vacant fields there are in that direction, and which is the piece blocking, if any. Thus, we decide that the board must have a function that can analyze a list of runs, and that the result is concatenated into a single list of vacant squares and a single list of neighboring pieces, if any. This function we call `getVacentNNeighbours`. And so we arrive at Listing 16.11.

Listing 16.11 pieces.fs:
An extension of chess base.

```
module Pieces
open Chess
/// A king moves 1 square in any direction
type king(col : Color) =
  inherit chessPiece(col)
  // A king has runs of length 1 in 8 directions:
  //   (N, NE, E, SE, S, SW, W, NW)
  override this.candidateRelativeMoves =
      [[(-1,0)];[(-1,1)];[(0,1)];[(1,1)];
      [(1,0)];[(1,-1)];[(0,-1)];[(-1,-1)]]
  override this.nameOfType = "king"
/// A rook moves horizontally and vertically
type rook(col : Color) =
  inherit chessPiece(col)
  // A rook can move horizontally and vertically
  // Make a list of relative coordinate lists. We consider the
  // current position and try all combinations of relative
  // moves (1,0); (2,0) ... (7,0); (-1,0); (-2,0); ...;
   (0,-7).
  // Some will be out of board, but will be assumed removed as
  // illegal moves.
  // A list of functions for relative moves
  let indToRel = [
    fun elm -> (elm,0); // South by elm
    fun elm -> (-elm,0); // North by elm
    fun elm -> (0,elm); // West by elm
    fun elm -> (0,-elm) // East by elm
    ]
  // For each function f in indToRel, we calculate
  //   List.map f [1..7].
```

The king has the simplest relative movement candidates, being the hypothetical eight neighboring squares. Rooks have a considerably longer list of candidates of relative moves, since it potentially can move to all 7 squares northward, eastward, southward, and westward. This could be hardcoded as 4 potential runs, `[[(1,0); (2,0); ... (7,0)]; [(-1,0); (-2,0); ... (0,-7)]]`. Each run will be based on the list `[1..7]`, which gives us the idea to use `List.map` to convert a list of single indices `[1..7]` into lists of runs as required by `candidateRelativeMoves`. Each run may be generated from `[1..7]` as

```
South: List.map (fun elm -> ( elm, 0)) [1..7]
North: List.map (fun elm -> (-elm, 0)) [1..7]
West:  List.map (fun elm -> (0,  elm)) [1..7]
East:  List.map (fun elm -> (0, -elm)) [1..7]
```

and which can be combined as a list of 4 lists of runs. Further, since functions are values, we can combine the 4 different anonymous functions into a list of functions and use a for-loop to iterate over the list of functions. This is shown in Listing 16.12. However, this solution is imperative in nature and does not use the elegance of the

Listing 16.12 imperativeRuns.fsx:
Calculating the runs of a rook using imperative programming.

```
30 let mutable listOfRuns : ((int * int) list) list = []
31 for f in indToRel do
32   let run = List.map f [1..7]
33   listOfRuns <- run :: listOfRuns
```

functional programming paradigm. A direct translation into functional programming is given in Listing 16.13. The functional version is slightly longer, but avoids the

Listing 16.13 functionalRuns.fsx:
Calculating the runs of a rook using functional programming.

```
30 let rec makeRuns lst =
31   match lst with
32     | [] -> []
33     | f :: rest ->
34       let run = List.map f [1..7]
35       run :: makeRuns rest
36 makeRuns indToRel
```

mutable variable.

Generating lists of runs from the two lists `[1..7]` and `indToRel` can also be performed with two `List.maps`, as shown in Listing 16.14.

The anonumous function,

```
fun e -> List.map e [1..7],
```

Listing 16.14 ListMapRuns.fsx:
Calculating the runs of a rook using double `List.maps`.

```
30 List.map (fun e -> List.map e [1..7]) indToRel
```

is used to wrap the inner `List.map` functional. An alternative, sometimes seen is to use currying with argument swapping: Consider the function, `let altMap lst e = List.map e lst`, which reverses the arguments of `List.map`. With this, the anonymous function can be written as `fun e -> altMap [1..7] e` or simply replaced by currying as `altMap [1..7]`. Reversing orders of arguments like this in combination with currrying is what the *`swap`* function is for,

```
let swap f a b = f b a.
```

With swap we can write `let altMap = swap List.map`. Thus,

```
swap List.map [1..7]
```

is the same function as `fun e -> List.map e [1..7]`, and in which case we could rewrite the solution in Listing 16.14 as

```
List.map (swap List.map [1..7]) indToRel
```

if we wanted a very compact, but possible less readable solution.

The final step will be to design the `Board` and `chessPiece` classes. The Chess module implements discriminated unions for color and an integer tuple for a position. These are shown in Listing 16.15. The `chessPiece` will need to know what a board

Listing 16.15 chess.fs:
A chess base: Module header and discriminated union types.

```
1 module Chess
2 type Color = White | Black
3 type Position = int * int
```

is, so we must define it as a mutually recursive class with `Board`. Furthermore, since all pieces must supply an implementation of `availableMoves`, we set it to be abstract by the abstract class attribute and with an abstract member. The board will need to be able to ask for a string describing each piece, and to keep the board on the screen we include an abbreviated description of the piece's properties color and piece type. The result is shown in Listing 16.16.

Our `Board` class is by far the largest and will be discussed in Listing 16.17–16.19. The constructor is shown in Listing 16.17. For memory efficiency, the board has been

Listing 16.16 chess.fs:
A chess base. Abstract type chessPiece.

```
4  /// An abstract chess piece
5  [<AbstractClass>]
6  type chessPiece(color : Color) =
7    let mutable _position : Position option = None
8    abstract member nameOfType : string // "king", "rook", ...
9    member this.color = color // White, Black
10   member this.position // E.g., (0,0), (3,4), etc.
11     with get() = _position
12     and set(pos) = _position <- pos
13   override this.ToString () = // E.g. "K" for white king
14     match color with
15       White -> (string this.nameOfType.[0]).ToUpper ()
16       | Black -> (string this.nameOfType.[0]).ToLower ()
17   /// A list of runs, which is a list of relative movements,
      e.g.,
18   /// [[(1,0); (2,0);...]; [(-1,0); (-2,0)]...]. Runs must be
19   /// ordered such that the first in a list is closest to the
      piece
20   /// at hand.
21   abstract member candidateRelativeMoves : Position list list
22   /// Available moves and neighbours ([(1,0); (2,0);...],
      [p1; p2])
23   member this.availableMoves (board : Board) : (Position list
      * chessPiece list) =
24     board.getVacantNNeighbours this
```

implemented using a `Array2D`, since pieces will move around often. For later use, in the members shown in Listing 16.19 we define two functions that convert relative coordinates into absolute coordinates on the board, and remove those that fall outside the board. These are called `validPositionWrap` and `relativeToAbsolute`.

For ease of use in an application, `Board` implements `Item`, such that the board can be read and written to using array notation. And `ToString` is overridden, such that an application may print the board anytime using a `printf` function. This is shown in Listing 16.18. Note that for efficiency, location is also stored in each piece, so `set` also needs to update the particular piece's position, as done in line 48. Note also that the board is printed with the first coordinate of the board being rows and second columns, and such that element (0,0) is at the bottom right complying with standard chess notation.

Listing 16.17 chess.fs:
A chess base: the constructor

```
25 /// A board
26 and Board () =
27   let _board = Collections.Array2D.create<chessPiece option>
      8 8 None
28   /// Wrap a position as option type
29   let validPositionWrap (pos : Position) : Position option =
30     let (rank, file) = pos // square coordinate
31     if rank < 0 || rank > 7 || file < 0 || file > 7 then
32       None
33     else
34       Some (rank, file)
35   /// Convert relative coordinates to absolute and remove
36   /// out-of-board coordinates.
37   let relativeToAbsolute (pos : Position) (lst : Position
      list) : Position list =
38     let addPair (a : int, b : int) (c : int, d : int) :
      Position =
39       (a+c,b+d)
40     // Add origin and delta positions
41     List.map (addPair pos) lst
42     // Choose absolute positions that are on the board
43     |> List.choose validPositionWrap
```

Listing 16.18 chess.fs:
A chess base: Board header, constructor, and non-static members.

```
44  /// Board is indexed using .[,] notation
45  member this.Item
46    with get(a : int, b : int) = _board.[a, b]
47    and set(a : int, b : int) (p : chessPiece option) =
48      if p.IsSome then p.Value.position <- Some (a,b)
49      _board.[a, b] <- p
50  /// Produce string of board for, e.g., the printfn function.
51  override this.ToString() =
52    let mutable str = ""
53    for i = Array2D.length1 _board - 1 downto 0 do
54      str <- str + string i
55      for j = 0 to Array2D.length2 _board - 1 do
56        let p =  _board.[i,j]
57        let pieceStr =
58            match p with
59              None -> " ";
60              | Some p -> p.ToString()
61        str <- str + " " + pieceStr
62      str <- str + "\n"
63    str + "  0 1 2 3 4 5 6 7"
64
65  /// Move piece by specifying source and target coordinates
66  member this.move (source : Position) (target : Position) :
     unit =
67    this.[fst target, snd target] <- this.[fst source, snd
     source]
68    this.[fst source, snd source] <- None
69  /// Find the tuple of empty squares and first neighbour if
     any.
70  member this.getVacantNOccupied (run : Position list) :
     (Position list * (chessPiece option)) =
71    try
72      // Find index of first non-vacant square of a run
73      let idx = List.findIndex (fun (i, j) ->
    this.[i,j].IsSome) run
```

The main computations are done in the static methods of the board, as shown in Listing 16.19. A chess piece must implement `candidateRelativeMoves`, and we de-

Listing 16.19 chess.fs:
A chess base: Board static members.

```
74        let (i,j) = run.[idx]
75        let piece = this.[i, j] // The first non-vacant
     neighbour
76        if idx = 0 then
77          ([], piece)
78        else
79          (run.[..(idx-1)], piece)
80      with
81        _ -> (run, None) // outside the board
82    /// find the list of all empty squares and list of
     neighbours
83    member this.getVacantNNeighbours (piece : chessPiece) :
     (Position list * chessPiece list)  =
84      match piece.position with
85        None ->
86          ([],[])
87        | Some p ->
88          let convertNWrap =
89            (relativeToAbsolute p) >> this.getVacantNOccupied
90          let vacantPieceLists = List.map convertNWrap
     piece.candidateRelativeMoves
91          // Extract and merge lists of vacant squares
92          let vacant = List.collect fst vacantPieceLists
93          // Extract and merge lists of first obstruction pieces
94          let neighbours = List.choose snd vacantPieceLists
95          (vacant, neighbours)
```

cided in Listing 16.16 that moves should be specified relative to the piece's position. Since the piece does not know which other pieces are on the board, it can only specify all potential positions. For convenience, we will allow pieces to also specify positions outside the board, such that, e.g., the rook can specify the 7 nearest neighboring squares up, down, left, and right, regardless that some may be outside the board. Thus `getVacantNNeighbours` must first convert the relative positions to absolute and clip any outside the board. This is done by `relativeToAbsolute`. Then for each run, the first occupied square must be identified. Since `availableMoves` must return two lists, vacant squares, and immediate neighbors, this structure is imposed on the output of `convertNWrap` as well. This is computed in `getVacantNOccupied` by use of the built-in `List.findIndex` function. This function returns the index of the first element in a list for which the supplied function is true and otherwise throws an exception. Exceptions are always somewhat inelegant, but in this case, it is harmless, since the exception signifies a valid situation where no pieces exist on the run. After having analyzed all runs independently, then all the vacant lists are merged, all the neighboring pieces are merged and both are returned to the caller.

Compiling the library files with the application and executing gives the result shown in Listing 16.20. We see that the program has correctly determined that initially, the

Listing 16.20: Running the program. Compare with Figure 16.3.

```
$ dotnet fsi chess.fs pieces.fs chessApp.fsx
7
6
5
4   k
3
2
1   R
0 K
  0 1 2 3 4 5 6 7
K: Some (0, 0) ([(0, 1); (1, 0)], [R])
R: Some (1, 1) ([(2, 1); (3, 1); (0, 1); (1, 2); (1, 3); (1,
   4); (1, 5); (1, 6); (1, 7); (1, 0)],
 [k])
k: Some (4, 1) ([(3, 1); (3, 2); (4, 2); (5, 2); (5, 1); (5,
   0); (4, 0); (3, 0)], [])
7
6
5
4   k
3   R
2
1
0 K
  0 1 2 3 4 5 6 7
K: Some (0, 0) ([(0, 1); (1, 1); (1, 0)], [])
R: Some (3, 1) ([(2, 1); (1, 1); (0, 1); (3, 2); (3, 3); (3,
   4); (3, 5); (3, 6); (3, 7); (3, 0)],
 [k])
k: Some (4, 1) ([(3, 2); (4, 2); (5, 2); (5, 1); (5, 0); (4,
   0); (3, 0)], [R])
```

white king has the white rook as its neighbors and due to its location in the corner only has two free positions to move to. The white rook has many and the black king as its neighbor. The black king is free to move to all its eight neighboring fields. After moving the white rook to (3,1) or b4 in regular chess notation, then the white king has no neighbors, and the white rook and the black king are now neighbors with an appropriate restriction on their respective vacant squares. These simple use-tests are in no way a thorough test of the quality of the code, but they give us a good indication that our library offers a tolerable interface for the application, and that at least major parts of the code function as expected. Thus, we conclude this intermezzo.

16.6 Key Concepts and Terms in This Chapter

In this chapter, we have looked at additional concepts from object-oriented programming focussing on class hierarchies. You have learned:

- how to create a new class by **inheriting** from another
- how to use **up-casting** and **down-casting** to mix objects of similar inheritance
- how to define abstract data types as **abstract classes**
- how to work with **class interfaces** to make objects compatible with generic functions

Chapter 17

Object-Oriented Design

Abstract Designing large programs is a challenging task. A key element of success is the encapsulation and definition of semantic meaningful units and interfaces. The object-oriented programming paradigm is supported by well-developed design paradigms. Here, we will examine both the *Universal Modelling Language* (*UML*), which is a graphical language for structuring object-oriented programs, and the *nouns and verbs* method for analyzing problem descriptions for candidates for classes as their interactions. In this chapter, you will learn how to

- use UML to visualize classes, their hierarchies, and their objects
- use the nouns and verbs method to analyze a problem statement or a use-story to identify potential candidates for classes and their interactions

Object-oriented programming has a well-developed methodology for analysis and design. The analysis serves as input to the design phase, where the analysis reveals *what* a program is supposed to do, and the design *how* it is supposed to be doing it. The analysis should be expressed in general terms irrespective of the technologic constraints, while the design should include technological constraints such as defined by the targeted language and hardware.

The primary steps for *object-oriented analysis and design* are:

1. identify objects,
2. describe object behavior,
3. describe object interactions,
4. describe some details of the object's inner workings,
5. write a precise description for classes, properties and methods using, e.g., F#'s XML documentation standard,
6. write mockup code,
7. write unit tests and test the basic framework using the mockup code,
8. replace the mockup with real code while testing to keep track of your progress. Extend the unit test as needed,
9. evaluate code in relation to the desired goal,
10. complete your documentation both in-code and outside.

Steps 1–4 are the analysis phase which gradually stops in step 4, while the design phase gradually starts at step 4 and gradually stops when actual code is written in step 7. Programming is never a linear experience, and you will often need to go back to previous steps to update or change decisions. You should not refrain from improving your program design and implementation, but you should always be mindful of the goal. Often less than the perfect solution will suffice.

An object-oriented analysis can be a daunting process. A good starting point is a *use case*, *problem statement*, or a *user story*, which in human language describes a number of possibly hypothetical interactions between a user and a system with the purpose of solving some task. Two useful methodologies for performing an object-oriented analysis is the method of nouns-and-verbs and the unified modeling language, described in the following sections.

17.1 Identification of Objects, Behaviors, and Interactions by Nouns-and-Verbs

Analyzing a specific use-case, good candidates for objects are persons, places, things, events, concept etc., which are almost always characterized by being *nouns* in the text. Interactions between objects are actions that bind objects together, and actions are often associated with *verbs*. When choosing methods, it is important to maintain an object-centered perspective, i.e., for a general-purpose object, we should limit the need for including information about other objects. E.g., a value of type `int` need not know anything about the program in which it is being used.

Said briefly, the *nouns-and-verbs method* is:

> Nouns are object candidates, and verbs are candidate methods that describe interactions between objects.

17.2 Class Diagrams in the Unified Modelling Language

Having found an initial list of candidate objects and interactions, it is often useful to make a drawing of these relations with an increased focus on the object's inner workings. A *class diagram* is a schematic drawing of the program, highlighting its object-oriented structure, and we will use the *Unified Modelling Language 2* (*UML*) [5] standard. The standard is very broad, and here we will discuss structure diagrams for use in describing objects.

A class is drawn as shown in Figure 17.1. In UML, classes are represented as

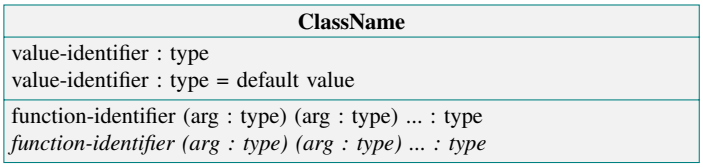


Fig. 17.1 A UML diagram for a class consists of it's name, zero or more attributes, and zero or more methods.

boxes with their class name. Depending on the desired level of details, zero or more properties and methods are described. These describe the basic interface to the class and objects of its type. Abstract members that require an implementation are shown in cursive. Here we have used F# syntax to conform with this book theme, but typically C# syntax is used. Interfaces are a special type of class that require an implementation. To highlight this, UML uses the notation shown in Figure 17.2.

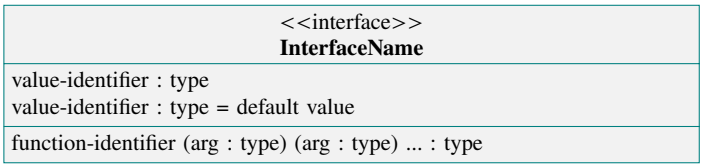


Fig. 17.2 An interface is a class that requires an implementation.

Relations between classes and objects are indicated by lines and arrows. The most common ones are summarized in Figure 17.3. Their meaning will be described in

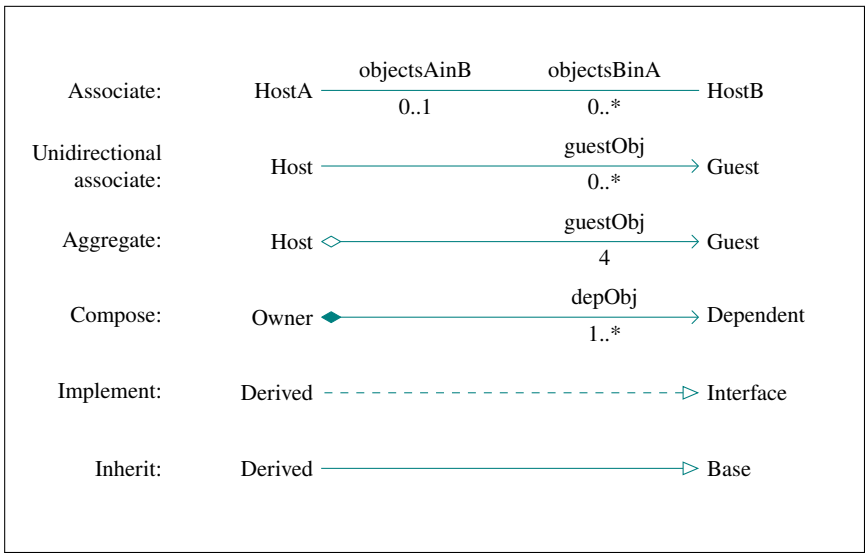


Fig. 17.3 Arrows used in class diagrams to show relations between objects.

detail in the following.

17.2.1 Associations

A family of relations is *association*, *aggregation*, and *composition*, and these are distinguished by how they handle the objects they are in relation with. The relation between the three relations is shown in Figure 17.4. Aggregational and compositional are specialized types of associations that imply ownership and are often called *has-a* relations. A composition is a collection of parts that makes up a whole. In object-oriented design, a compositional relation is a strong relation, where a guest object makes little sense without the host, as a room cannot exist without a house. An aggregation is a collection of assorted items, and in object-oriented design, an aggregational relation is a loose relation, like how a battery can meaningfully

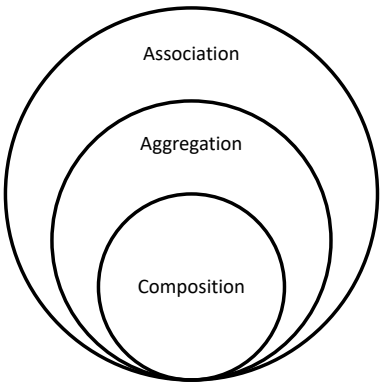


Fig. 17.4 The relation between Association, Aggregation and Composition in UML.

be separated from a torchlight. Some associations are neither aggregational nor compositional, and commonly just called an association. An association is a group of people or things linked for some common purpose a cooccurrence. In object-oriented design, associations between objects are the loosest possible relations, like how a student may be associated with the local coffee shop. Sometimes associational relations are called a *knows-about*.

The most general type of association, which is just called an association, is the possibility for objects to send messages to each other. This implies that one class knows about the other, e.g., uses it as arguments of a function or similar. A host is associated with a guest if the host has a reference to the guest. Objects are reference types, and therefore, any object which is not created by the host, but where a name is bound to a guest object but not explicitly copied, then this is an association relation.

Bidirectional association means that classes know about each other. The UML notation is shown in Figure 17.5. Association may be annotated by an identifier and

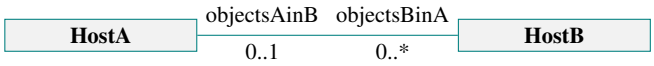


Fig. 17.5 Bidirectional association is shown as a line with optional annotation.

a multiplicity. In the figure, HostA has 0 or more variables of type HostB named objectsBinA, while HostB has 0 or 1 variables of HostA named objectsAinB. The multiplicity notation is very similar to F#'s slicing notation. Typical values are shown in Table 17.1. If the association is unidirectional, then an arrow is added for empha-

n	exactly n instances
*	zero or more instances
n..m	n to m instances
n..*	from n to infinite instances

Table 17.1 Notation for association multiplicities is similar to F#'s slicing notation.

sis, as shown in Figure 17.6. In this example, Host knows about Guest and has one instance of it, and Guest is oblivious about Host.

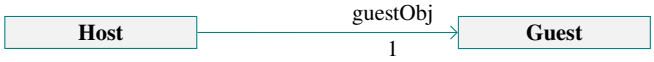


Fig. 17.6 Unidirectional association shows a one-side *has-a* relation.

A programming example showing a unidirectional association is given in Listing 17.1. Here, the `student` is unidirectionally associated with a `teacher` since the

Listing 17.1 umlAssociation.fsx:
The `student` is associated with a `teacher`.

```
type teacher () =
  member this.answer (q : string) = "4"
type student (t : teacher) =
  member this.ask () = t.answer("What is 2+2?")

let t = teacher ()
let s = student (t)
s.ask()
```

`student` can send and receive messages to and from the `teacher`. The `teacher`, on the other hand, does not know anything about the `student`. In UML this is depicted as shown in Figure 17.7.

Fig. 17.7 The `teacher` and `student` objects can access each other's functions, and thus they have an association relation.

Aggregated relationships are a specialization of associations. As an example, an author may have written a book, but once created, the book gets a life independent of the author and may, for example, be given to a reader, and the book continues to exist even when the author dies. That is, In aggregated relations, the host object has a reference to a guest object and may have created the guest, but the guest will be shared with other objects, and when the host is deleted, the guest is not.

Aggregation is illustrated using a diamond tail and an open arrow, as shown in Figure 17.8. Here the Host class has stored aliases to four different Guest objects.

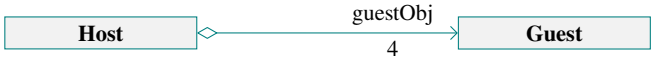


Fig. 17.8 Aggregation relations are a subset of associations where local aliases are stored for later use.

An programming example of an aggregation relation is given in Listing 17.2. In aggregated relations, there is a sense of ownership, and in the example, the `author` object creates a `book` object which is published and bought by a reader. Hence the `book` change ownership during the execution of the program. In UML this is to be depicted as shown in Figure 17.9.

Listing 17.2 umlAggregation.fsx:
The book has an aggregated relation to `author` and `reader`.

```
type book (name : string) =
  let mutable _name = name
type author () =
  let _book = book("Learning to program")
  member this.publish() = _book
type reader () =
  let mutable _book : book option = None
  member this.buy (b : book) = _book <- Some b

let a = author ()
let r = reader ()
let b = a.publish ()
r.buy (b)
```

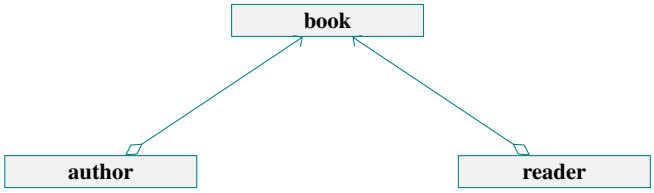


Fig. 17.9 A book is an object that can be owned by both an author and a reader.

A compositional relationship is a specialization of aggregations. As an example, a dog has legs, and dog legs can not very sensibly be given to other animals. That is, in compositional relations, the host creates the guest, and when the host is deleted, so is the guest. A composition is a stronger relation than aggregation and is illustated using a filled filled diamond tail, as illustrated in Figure 17.10. In this example,

Fig. 17.10 Composition relations are a subset of aggregation where the host controls the lifetime of the guest objects.

Owner has created 1 or more objects of type Dependent, and when Owner is deleted, so are these objects.

A programming example of a composition relation is given in Listing 17.3. In

Listing 17.3 umlComposition.fsx:
The dog object is a composition of four `leg` objects.

```
type leg () =
  member this.move = "moving"
type dog () =
  let _leg = List.init 4 (fun e -> leg ())

let bestFriend = dog ()
```

Listing 17.3, a `dog` object creates four `leg` objects, and it makes less sense to be able to turn over the ownership of each `leg` to other objects. Thus, a `dog` is a composition of `leg` objects. Using UML, this should be depicted as shown in Figure 17.11.

Fig. 17.11 A dog is a composition of legs.

17.2.2 Inheritance-type relations

Classes may inherit other classes where the parent is called the base class and the children its derived classes. Such a relation is often called an *is-a* relation, since the derived class *is a* kind of base class.

Inheritance is a relation between properties of classes. As an example, a student and a teacher is a type of person. All persons have names, while a student also has a reading list, and a teacher also has a a set of slides. Thus, both students and teacher may inherit from a person to gain the common property, name. In UML this is illustated with an non-filled, closed arrow as shown in Figure 17.12. Here two

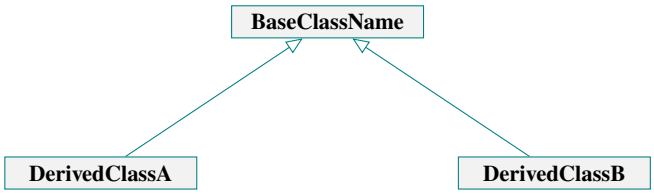


Fig. 17.12 Inheritance is shown by a closed arrowhead pointing to the base.

classes inherit the base class.

A programming example of an inheritance is given in Listing 17.4. In Listing 17.4, the `student` and the `teacher` classes are derived from the same `person` class. Thus, they all three have the `name` property. Using UML, this should be depicted as shown in Figure 17.13.

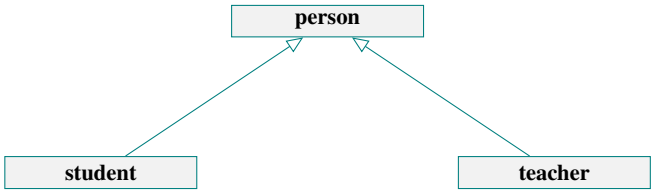


Fig. 17.13 A student and a teacher inherit from a person class.

Listing 17.4 umlInheritance.fsx:
The `student` and the `teacher` class inherits from the `person` class.

```
type person (name : string) =
  member this.name = name
type student (name : string, book : string) =
  inherit person(name)
  member this.book = book
type teacher (name : string, slides : string) =
  inherit person(name)
  member this.slides = slides

let s = student("Hans", "Learning to Program")
let t = teacher("Jon", "Slides of the day")
```

An interface is a relation between the properties of an abstract class and a regular class. As an example, a television and a car both have buttons, that you can press, although their effect will be quite different. Thus, a television and a car may both implement the same interface. In UML, interfaces are shown similarly to inheritance, but using a stippled line, as shown in Figure 17.14.

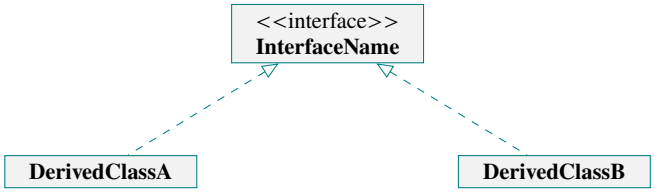


Fig. 17.14 Implementations of interfaces is shown with stippled line and closed arrowhead pointing to the base.

A programming example of an interface is given in Listing 17.5. In Listing 17.5, the

Listing 17.5 umlInterface.fsx:
The `television` and the `car` class both implement the `button` interface.

```
type button =
  abstract member press : unit -> string
type television () =
  interface button with
    member this.press () = "Changing channel"
type car () =
  interface button with
    member this.press () = "Activating wipers"
let pressIt (elm : #button) =
  elm.press()

let t = television()
let c = car()
printfn "%s" (pressIt t)
printfn "%s" (pressIt c)
```

`television` and the `car` classes implement the `button` interface. Hence, although they are different classes, they both have the `press ()` method and, e.g., can be given as a function requiring only the existence of the `press ()` method. Using UML, this should be depicted as shown in Figure 17.15.

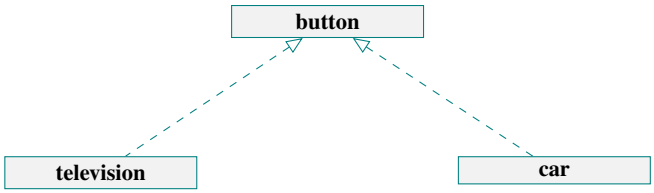


Fig. 17.15 A student and a teacher inherit from a person class.

17.2.3 Packages

For visual flair, modules and namespaces are often visualized as *packages*, as shown in Figure 17.16. A package is like a module in F#.

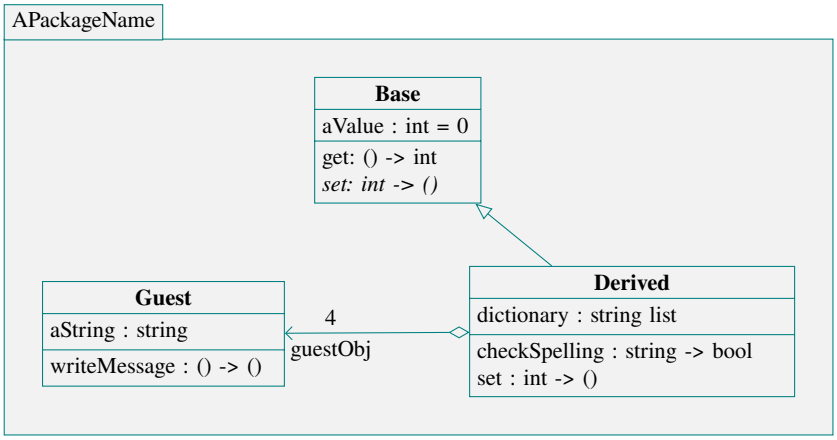


Fig. 17.16 Packages are a visualizations of modules and namespaces.

17.3 Programming Intermezzo: Designing a Racing Game

An example is the following *problem statement*:

Problem 17.1

rite a racing game, where each player controls his or her vehicle on a track. Each vehicle must have individual features such as top acceleration, speed, and handling. The player must be able to turn the vehicle left and right, and to accelerate up and down. At the beginning of the game, each vehicle is placed behind the starting line. Once the start signal is given, then the players may start to operate their vehicles. The player who first completes 3 rounds wins.

To seek a solution, we will use the **nouns**-and-verbs method. Below, the problem statement is repeated with **nouns** and verbs highlighted.

> Write a **racing game**, where each **player** controls his or her **vehicle** on a **track**. Each **vehicle** must have individual **features** such as **top acceleration**, **speed**, and **handling**. The **player** must be able to turn the vehicle left and right, and to accelerate up and down. At the **beginning** of the **game**, each **vehicle** is placed behind the **starting line**. Once the **start signal** is given, then the **players** may start to operate their **vehicles**. The **player** who first completes 3 **rounds** wins.

The above nouns and verbs are candidates for objects, their behaviour, and their interaction. A deeper analysis is:

Identification of objects by nouns (Step 1):

Identified unique nouns are: **racing game (game)**, **player**, **vehicle**, **track**, **feature**, **top acceleration**, **speed**, **handling**, **beginning**, **starting line**, **start signal**, **rounds**. From this list we seek cohesive units that are independent and reusable. The nouns

> **game**, **player**, **vehicle**, and **track**

seem to fulfill these requirements, while all the rest seems to be features of the former and thus not independent concepts. E.g., **top acceleration** is a feature of a **vehicle**, and **starting line** is a feature of a **track**.

Object behavior and interactions by verbs (Steps 2 and 3):

To continue our object-oriented analysis, we will consider the object candidates identified above, and verbalize how they would act as models of general concepts useful in our game.

player The **player** is associated with the following verbs:

- A **player** controls/operates a **vehicle**.
- A **player** turns and accelerates a **vehicle**.
- A **player** completes **rounds**.

- A **player** wins.

Verbalizing a **player**, we say that a **player** in general must be able to control the **vehicle**. In order to do this, the **player** must receive information about the **track** and all **vehicles**, or at least some information about the nearby **vehicles** and **track**. Furthermore, the **player** must receive information about the state of the **game**, i.e., when the race starts and stops.

vehicle A **vehicle** is controlled by a **player** and further associated with the following verbs:

- A **vehicle** has **features top acceleration**, **speed**, and **handling**.
- A **vehicle** is placed on the **track**.

To further describe a **vehicle**, we say that a **vehicle** is a model of a physical object which moves around on the **track** under the influence of a **player**. A **vehicle** must have a number of attributes such as top acceleration, speed, and handling, and must be able to receive information about when to turn and accelerate. A **vehicle** must be able to determine its location in particular if it is on or off **track** and, and it must be able to determine if it has crashed into an obstacle such as another **vehicle**.

track A **track** is the place where vehicles operate and is further associated with the following verbs:

- A **track** has a **starting line**.
- A **track** has rounds.

Thus, a **track** is a fixed entity on which the **vehicles** race. It has a size and a shape, a starting and a finishing line, which may be the same, and **vehicles** may be placed on the **track** and can move on and possibly off the **track**.

game Finally, a **game** is associated with the following verbs:

- A **game** has a **beginning** and a **start signal**.
- A **game** can be **won**.

A **game** is the total sum of all the **players**, the **vehicles**, the **tracks**, and their interactions. A **game** controls events, including inviting **players** to race, sending the **start signal**, and monitoring when a **game** is finished and who **won**.

From the above we see that the object candidates **features** seems to be a natural part of the description of the **vehicle**'s attributes, and similarly, a **starting**

line may be an intricate part of a **track**. Also, many of the *verbs* used in the problem statement and in our extended verbalization of the general concepts indicate methods that are used to interact with the object. The object-centered perspective tells us that for a general-purpose **vehicle** object, we need not include information about the **player**, analogous to how a value of type `int` need not know anything about the program, in which it is being used. In contrast, the candidate **game** is not as easily dismissed and could be used as a class which contains all the above.

With this description, we see that 'start signal' can be included as a natural part of the game object. Being confident in our working hypothesis of the essential objects for the solution, we continue our investigation into further details about the objects and their interactions.

Analysis details (Step 4):

A class diagram of our design for the proposed classes and their relations is shown in Figure 17.17.

In the present description, there will be a single Game object that initializes the other objects, executes a loop updating the clock, queries the players for actions, and informs the vehicles that they should move and under what circumstances. The track has been chosen to be dumb and does not participate much in the action. Player's method getAction will be an input from a user by keyboard, joystick or similar, but the complexity of the code for a computer player will be large, since it needs to take a sensible decision based on the track and the location of the other vehicles. What at present is less clear, is whether it is the responsibility of Game or Vehicle to detect an off track or a crash event. If a vehicle is to do this, then each vehicle must have aggregated association to all other vehicles and obstacles. So, on the one hand, it would seem an elegant delegation of responsibilities that a vehicle knows whether it has crashed into an obstacle or not, but on the other hand, it seems wasteful of memory resources to have duplicated references of all obstacles in every vehicle. The final choice is thus one of elegance versus resource management, and in the above, we have favored resource management. Thus, the main loop in Game must check all vehicles for a crash event after the vehicle's positions have been updated, and in case of a crash, informs the relevant vehicles.

Having created a design for a racing game, we are now ready to start coding (Step 6–). It is not uncommon that transforming our design into code will reveal new structures and problems that possibly require our design to be updated. Nevertheless, a good design phase is almost always a sure course to avoid many problems once coding, since the design phase allows the programmer to think about the problem from a helicopter perspective before tackling details of specific sub-problems.

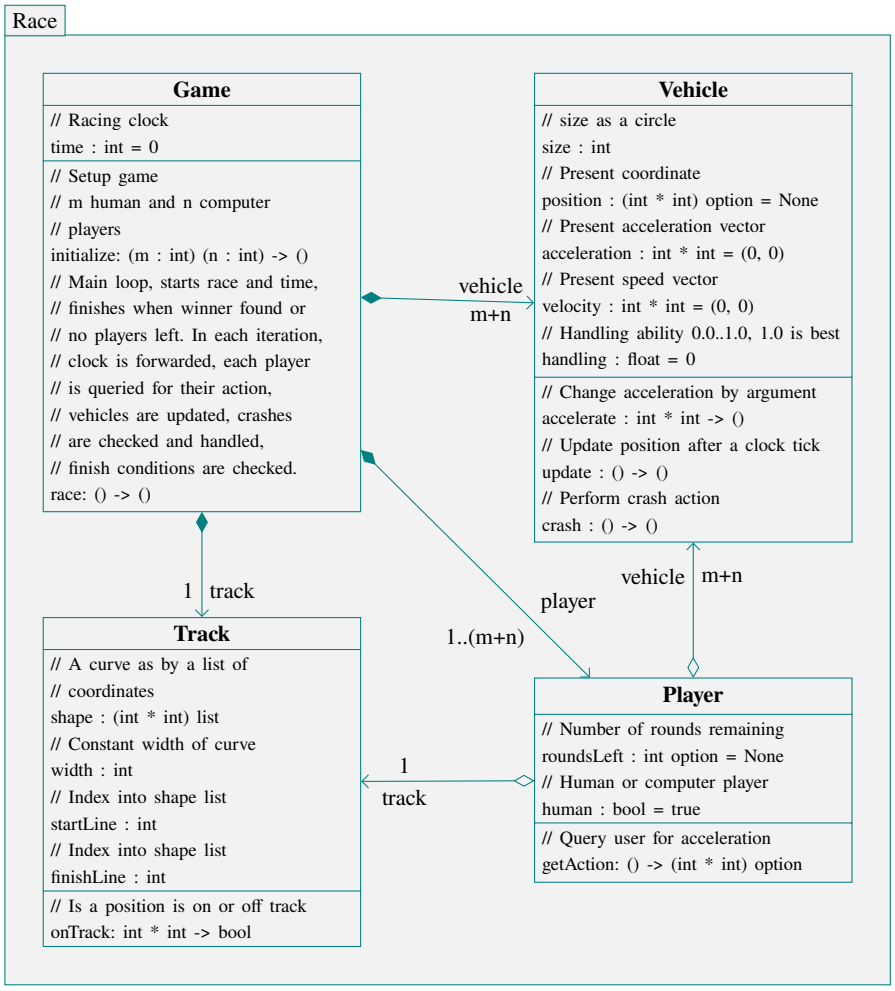


Fig. 17.17 A class diagram for a racing game.

17.4 Key Concepts and Terms in This Chapter

In this chapter, we have looked at object-oriented design. Two key elements have been discussed:

- how to organize and reason about classes and objects and their interactions using **UML** diagrams
- how in UML classes and objects are drawn as boxes and their relations are shown as lines with varying arrow-heads and -tails to denote the relation kind

- that class relations can be described as a **composition**, an **aggregation**, and an **association**
- that classes in an inheritance relation can be called an **is-a** relation, while aggregational and compositional relations can be called an **has-a** relation

Part IV

Appendices

Appendix A

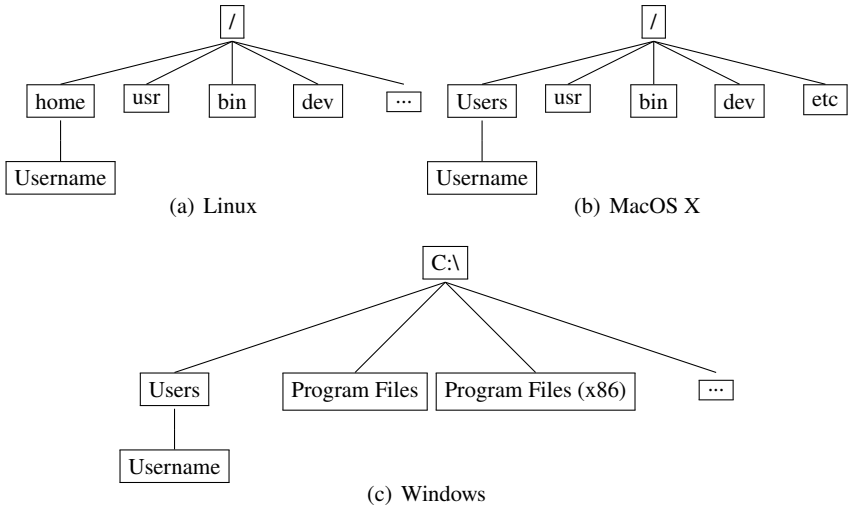
The Console in Windows, MacOS X, and Linux

Almost all popular operating systems are accessed through a user-friendly *graphical user interface* (*GUI*) that is designed to make typical tasks easy to learn to solve. As a computer programmer, you often need to access some of the functionalities of the computer, which, unfortunately, are sometimes complicated by this particular graphical user interface. The *console*, also called the *terminal* and the *Windows command line*, is the right hand of a programmer. The console is a simple program that allows you to complete text commands. Almost all the tasks that can be done with the graphical user interface can be done in the console and vice versa. Using the console, you will benefit from its direct control of the programs we write, and in your education, you will benefit from the fast and raw information you get through the console.

A.1 The Basics

When you open a *directory* or *folder* in your preferred operating system, the directory will have a location in the file system, whether from the console or through the operating system's graphical user interface. The console will almost always be associated with a particular directory or folder in the file system, and it is said that it is the directory that the console is in. The exact structure of file systems varies between Linux, MacOS X, and Windows, but common is that it is a hierarchical structure. This is illustrated in Figure A.1.

There are many predefined console commands, available in the console, and you can also make your own. In the following sections, we will review the most important commands in the three different operating systems. These are summarized in Table A.1.



(a) Linux (b) MacOS X

(c) Windows

Fig. A.1 The top file hierarchy levels of common operating systems.

Windows	MacOS X/Linux	Description
`dir`	`ls`	Show content of present directory.
`cd <d>`	`cd <d>`	Change present directory to `<d>`.
`mkdir <d>`	`mkdir <d>`	Create directory `<d>`.
`rmdir <d>`	`rmdir <d>`	Delete `<d>` (Warning: cannot be reverted).
`move <f> <f | d>`	`mv <f> <f | d>`	Move `<fil>` to `<f | d>`.
`copy <f1> <f2>`	`cp <f1> <f2>`	Create a new file called `<f2>` as a copy of `<f1>`.
`del <f>`	`rm <f>`	delete `<f>` (Warning: cannot be reverted).
`echo <s | v>`	`echo <s | v>`	Write a string or content of a variable to screen.

Table A.1 The most important console commands for Windows, MacOS X, and Linux. Here `<f*>` is shorthand for any filename, `<d>` for any directory name, `<s>` for any string, and `<v>` for any shell-variable.

A.2 Windows

In this section we will discuss the commands summarized in Table A.1. Windows 7 and earlier versions: To open the console, press `Start->Run` in the lower left corner, and then type `cmd` in the box. In Windows 8 and 10, you right-click on the windows icon, choose `Run` or equivalent in your local language, and type `cmd`. Alternatively, you can type `Windows-key + R`. Now you should open a console window with a prompt showing something like Listing A.1.

Listing A.1: The Windows console.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights
   reserved.

C:\Users\sporring>
```

To see which files are in the directory, use *dir*, as shown in Listing A.2.

Listing A.2: Directory listing with dir.

```
C:\Users\sporring>dir
 Volume in drive C has no label.
 Volume Serial Number is 94F0-31BD

 Directory of C:\Users\sporring

30-07-2015  15:23    <DIR>          .
30-07-2015  15:23    <DIR>          ..
30-07-2015  14:27    <DIR>          Contacts
30-07-2015  14:27    <DIR>          Desktop
30-07-2015  17:40    <DIR>          Documents
30-07-2015  15:11    <DIR>          Downloads
30-07-2015  14:28    <DIR>          Favorites
30-07-2015  14:27    <DIR>          Links
30-07-2015  14:27    <DIR>          Music
30-07-2015  14:27    <DIR>          Pictures
30-07-2015  14:27    <DIR>          Saved Games
30-07-2015  17:27    <DIR>          Searches
30-07-2015  14:27    <DIR>          Videos
               0 File(s)              0 bytes
              13 Dir(s)  95.004.622.848 bytes free

C:\Users\sporring>
```

We see that there are no files and thirteen directories (DIR). The columns tell from left to right: the date and time of their creation, the file size or if it is a folder, and the name file or directory name. The first two folders "`.`" and "`..`" are found in each folder and refer to this folder as well as the one above in the hierarchy. In this case, the folder "`.`" is an alias for `C:\Users\sporring` and "`..`" for `C:\Users`.

Use *cd* to change directory, e.g., to `Documents`, as in Listing A.3.

Listing A.3: Change directory with cd.

```
C:\Users\sporring>cd Documents

C:\Users\sporring\Documents>
```

Note that some systems translate default filenames, so their names may be given different names in different languages in the graphical user interface as compared to the console.

You can use *mkdir* to create a new directory called, e.g., `myFolder`, as illustrated in Listing A.4.

Listing A.4: Creating a directory with `mkdir`.

```
C:\Users\sporring\Documents>mkdir myFolder

C:\Users\sporring\Documents>dir
 Volume in drive C has no label.
 Volume Serial Number is 94F0-31BD

 Directory of C:\Users\sporring\Documents

30-07-2015  19:17    <DIR>          .
30-07-2015  19:17    <DIR>          ..
30-07-2015  19:17    <DIR>          myFolder
               0 File(s)              0 bytes
               3 Dir(s)  94.656.638.976 bytes free

C:\Users\sporring\Documents>
```

By using `dir` we inspect the result.

Files can be created by, e.g., *echo* and *redirection*, as demonstrated in Listing A.5.

Listing A.5: Creating a file with `echo` and redirection.

```
C:\Users\sporring\Documents>echo "Hi" > hi.txt

C:\Users\sporring\Documents>dir
 Volume in drive C has no label.
 Volume Serial Number is 94F0-31BD

 Directory of C:\Users\sporring\Documents

30-07-2015  19:18    <DIR>          .
30-07-2015  19:18    <DIR>          ..
30-07-2015  19:17    <DIR>          myFolder
30-07-2015  19:18                 8 hi.txt
               1 File(s)              8 bytes
               3 Dir(s)  94.656.634.880 bytes free

C:\Users\sporring\Documents>
```

To move the file `hi.txt` to the directory `myFolder`, use *move*, as shown in Listing A.6.

Listing A.6: Move a file with `move`.

```
C:\Users\sporring\Documents>move hi.txt myFolder
        1 file(s) moved.

C:\Users\sporring\Documents>
```

Finally, use *`del`* to delete a file and *`rmdir`* to delete a directory, as shown in Listing A.7.

Listing A.7: Delete files and directories with `del` and `rmdir`.

```
C:\Users\sporring\Documents>cd myFolder

C:\Users\sporring\Documents\myFolder>del hi.txt

C:\Users\sporring\Documents\myFolder>cd ..

C:\Users\sporring\Documents>rmdir myFolder

C:\Users\sporring\Documents>dir
 Volume in drive C has no label.
 Volume Serial Number is 94F0-31BD

 Directory of C:\Users\sporring\Documents

30-07-2015  19:20    <DIR>          .
30-07-2015  19:20    <DIR>          ..
               0 File(s)              0 bytes
               2 Dir(s)  94.651.142.144 bytes free

C:\Users\sporring\Documents>
```

The commands available from the console must be in its *search path*. The search path can be seen using `echo`, as shown in Listing A.8.

Listing A.8: Displaying the search path.

```
C:\Users\sporring\Documents>echo %Path%
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;
   C:\Windows\System32\WindowsPowerShell\v1.0\;"\Program
   Files\emacs-24.5\bin\"

C:\Users\sporring\Documents>
```

The path can be changed using the Control panel in the graphical user interface. In Windows 7, choose the Control panel, choose `System and Security` → `System` → `Advanced system settings` → `Environment Variables`. In Windows 10, you can find this window by searching for “Environment” in the Control panel. In

the window's System variables box, double-click on Path and add or remove a path from the list. The search path is a list of paths separated by ";". Beware, Windows uses the search path for many different tasks, so remove only paths that you are certain are not used for anything.

A useful feature of the console is that you can use the tab-key to cycle through filenames. E.g., if you write cd followed by a space and tab a couple of times, then the console will suggest to you the available directories.

A.3 MacOS X and Linux

MacOS X (OSX) and Linux are very similar, and both have the option of using *bash* as console. It is in the standard console on MacOS X and on many Linux distributions. A summary of the most important bash commands is shown in Table A.1. In MacOS X, you find the console by opening Finder and navigating to Applications → Utilities -> Terminal. In Linux, the console can be started by typing Ctrl + Alt + T. Some Linux distributions have other key-combinations such as Super + T.

Once opened, the console is shown in a window with content, as shown in Listing A.9.

Listing A.9: The MacOS console.

```
Last login: Thu Jul 30 11:52:07 on ttys000
FN11194:~ sporring$
```

"FN11194" is the name of the computer, the character ~ is used as an alias for the user's home directory, and "sporring" is the username for the user presently logged onto the system. Use *ls* to see which files are present, as shown in Listing A.10.

Listing A.10: Display a directory content with ls.

```
FN11194:~ sporring$ ls
Applications    Documents    Library        Music
   Public
Desktop        Downloads    Movies        Pictures
FN11194:~ sporring$
```

More details about the files are available by using flags to ls as demonstrated in Listing A.11.

Listing A.11: Display extra information about files using flags to `ls`.

```
FN11194:~ sporring$ ls -l
drwx------   6 sporring  staff   204 Jul 30 14:07
   Applications
drwx------+ 32 sporring  staff  1088 Jul 30 14:34 Desktop
drwx------+ 76 sporring  staff  2584 Jul  2 15:53 Documents
drwx------+  4 sporring  staff   136 Jul 30 14:35 Downloads
drwx------@ 63 sporring  staff  2142 Jul 30 14:07 Library
drwx------+  3 sporring  staff   102 Jun 29 21:48 Movies
drwx------+  4 sporring  staff   136 Jul  4 17:40 Music
drwx------+  3 sporring  staff   102 Jun 29 21:48 Pictures
drwxr-xr-x+  5 sporring  staff   170 Jun 29 21:48 Public
FN11194:~ sporring$
```

The flag `-l` means long, and many other flags can be found by querying the built-in manual with `man ls`. The output is divided into columns, where the left column shows a number of codes: "d" stands for directory, and the set of three of optional "rwx" denote whether respectively the owner, the associated group of users, and anyone can respectively "r" - read, "w" - write, and "x" - execute the file. In all directories but the `Public` directory, only the owner can do any of the three. For directories, "x" means permission to enter. The second column can often be ignored, but shows how many links there are to the file or directory. Then follows the username of the owner, which in this case is `sporring`. The files are also associated with a group of users, and in this case, they all are associated with the group called `staff`. Then follows the file or directory size, the date of last change, and the file or directory name. There are always two hidden directories: "`.`" and "`..`", where "`.`" is an alias for the present directory, and "`..`" for the directory above. Hidden files will be shown with the `-a` flag.

Use *`cd`* to change to the directory, for example to `Documents` as shown in Listing A.12.

Listing A.12: Change directory with `cd`.

```
FN11194:~ sporring$ cd Documents/
FN11194:Documents sporring$
```

Note that some graphical user interfaces translate standard filenames and directories to the local language, such that navigating using the graphical user interface will reveal other files and directories, which, however, are aliases.

You can create a new directory using *`mkdir`*, as demonstrated in Listing A.13.

Listing A.13: Creating a directory using `mkdir`.

```
FN11194:Documents sporring$  mkdir myFolder
FN11194:Documents sporring$ ls
myFolder
FN11194:tmp sporring$
```

A file can be created using *echo* and with *redirection*, as shown in Listing A.14.

Listing A.14: Creating a file with `echo` and redirection.

```
FN11194:Documents sporring$ echo "hi" > hi.txt
FN11194:Documents sporring$ ls
hi.txt        myFolder
```

To move the file `hi.txt` into `myFolder`, use *mv*. This is demonstrated in Listing A.15.

Listing A.15: Moving files with `mv`.

```
FN11194:Documents sporring$ echo mv hi.txt myFolder/
FN11194:Documents sporring$
```

To delete the file and the directory, use *rm* and *rmdir*, as shown in Listing A.16.

Listing A.16: Deleting files and directories.

```
FN11194:Documents sporring$ cd myFolder/
FN11194:myFolder sporring$ rm hi.txt
FN11194:myFolder sporring$ cd ..
FN11194:Documents sporring$ rmdir myFolder/
FN11194:Documents sporring$ ls
FN11194:Documents sporring$
```

Only commands found on the *search path* are available in the console. The content of the search path is seen using the `echo` command, as demonstrated in Listing A.17.

Listing A.17: The content of the search path.

```
FN11194:Documents sporring$ echo $PATH
/Applications/Maple
   17/:/Applications/PackageMaker.app/Contents/MacOS/:
   /Applications/MATLAB_R2014b.app/bin/:/opt/local/bin:
   /opt/local/sbin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:
   /sbin:/opt/X11/bin:/Library/TeX/texbin
FN11194:Documents sporring$
```

The search path can be changed by editing the setup file for Bash. On MacOS X it is called `~/.profile`, and on Linux it is either `~/.bash_profile` or `~/.bashrc`.

Here new paths can be added by adding the following line: `export PATH="<new path>:<another new path>:$PATH"`.

A useful feature of Bash is that the console can help you write commands. E.g., if you write `fs` followed by pressing the `tab`-key, and if Mono is in the search path, then Bash will typically respond by completing the line as `fsharp`, and by further pressing the `tab`-key some times, Bash will show the list of options, typically `fshpari` and `fsharpc`. Also, most commands have an extensive manual which can be accessed using the `man` command. E.g., the manual for `rm` is retrieved by `man rm`.

Appendix B

Number Systems on the Computer

B.1 Binary Numbers

Humans like to use the *decimal number* system for representing numbers. Decimal numbers are *base* 10 meaning that a decimal number consists of a sequence of digits separated by a *decimal point*, where each *digit* can have values $d \in \{0, 1, 2, \ldots, 9\}$ and the weight of each digit is proportional to its place in the sequence of digits with respect to the decimal point, i.e., the number $357.6 = 3 \cdot 10^2 + 5 \cdot 10^1 + 7 \cdot 10^0 + 6 \cdot 10^{-1}$, or in general, for a number consisting of digits d_i with $n + 1$ and m digits to the left and right of the decimal point, the value v is calculated as:

$$v = \sum_{i=-m}^{n} d_i 10^i. \tag{B.1}$$

The basic unit of information in almost all computers is the binary digit, or *bit* for short. A *binary number* consists of a sequence of binary digits separated by a decimal point, where each digit can have values $b \in \{0, 1\}$, and the base is 2. The general equation is,

$$v = \sum_{i=-m}^{n} b_i 2^i, \tag{B.2}$$

and examples are $1011.1_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} = 11.5$. Notice that we use subscript 2 to denote a binary number, while no subscript is used for decimal numbers. The left-most bit is called the *most significant bit*, and the right-most bit is called the *least significant bit*. Due to typical organisation of computer memory, 8 binary digits is called a *byte*, and the term *word* is not universally defined but typically related to the computer architecture, a program is running on, such as 32 or 64 bits.

Other number systems are often used, e.g., *octal numbers*, which are base 8 numbers and have digits $o \in \{0, 1, \ldots, 7\}$. Octals are useful short-hand for binary, since 3 binary digits map to the set of octal digits. Likewise, *hexadecimal numbers* are base 16 with digits $h \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$, such that $a_{16} = 10$, $b_{16} = 11$ and so on. Hexadecimals are convenient, since 4 binary digits map directly to the set of hexadecimal digits. Thus $367 = 101101111_2 = 557_8 = 16f_{16}$. A list of the integers 0–63 in various bases is given in Table B.1.

Dec	Bin	Oct	Hex	Dec	Bin	Oct	Hex
0	0	0	0	32	100000	40	20
1	1	1	1	33	100001	41	21
2	10	2	2	34	100010	42	22
3	11	3	3	35	100011	43	23
4	100	4	4	36	100100	44	24
5	101	5	5	37	100101	45	25
6	110	6	6	38	100110	46	26
7	111	7	7	39	100111	47	27
8	1000	10	8	40	101000	50	28
9	1001	11	9	41	101001	51	29
10	1010	12	a	42	101010	52	2a
11	1011	13	b	43	101011	53	2b
12	1100	14	c	44	101100	54	2c
13	1101	15	d	45	101101	55	2d
14	1110	16	e	46	101110	56	2e
15	1111	17	f	47	101111	57	2f
16	10000	20	10	48	110000	60	30
17	10001	21	11	49	110001	61	31
18	10010	22	12	50	110010	62	32
19	10011	23	13	51	110011	63	33
20	10100	24	14	52	110100	64	34
21	10101	25	15	53	110101	65	35
22	10110	26	16	54	110110	66	36
23	10111	27	17	55	110111	67	37
24	11000	30	18	56	111000	70	38
25	11001	31	19	57	111001	71	39
26	11010	32	1a	58	111010	72	3a
27	11011	33	1b	59	111011	73	3b
28	11100	34	1c	60	111100	74	3c
29	11101	35	1d	61	111101	75	3d
30	11110	36	1e	62	111110	76	3e
31	11111	37	1f	63	111111	77	3f

Table B.1 A list of the integers 0–63 in decimal, binary, octal, and hexadecimal.

B.2 IEEE 754 Floating Point Standard

The set of real numbers, also called *reals*, includes all fractions and irrational numbers. It is infinite in size both in the sense that there is no largest nor smallest number,

and that between any 2 given numbers there are infinitely many numbers. Reals are widely used for calculation, but since any computer only has finite memory, there are infinitely many numbers which cannot be represent on a computer. Hence, any computation performed on a computer with reals must rely on approximations. *IEEE 754 double precision floating-point format* (*binary64*), known as a *double*, is a standard for representing an approximation of reals using 64 bits. These bits are divided into 3 parts: sign, exponent and fraction,

$$s\, e_1 e_2 \ldots e_{11}\, m_1 m_2 \ldots m_{52},$$

where s, e_i, and m_j are binary digits. The bits are converted to a number using the equation by first calculating the exponent e and the mantissa m,

$$e = \sum_{i=1}^{11} e_i 2^{11-i}, \tag{B.3}$$

$$m = \sum_{j=1}^{52} m_j 2^{-j}. \tag{B.4}$$

I.e., the exponent is an integer, where $0 \leq e < 2^{11}$, and the mantissa is a rational, where $0 \leq m < 1$. For most combinations of e and m, the real number v is calculated as,

$$v = (-1)^s\, (1 + m)\, 2^{e-1023} \tag{B.5}$$

with the exceptions that

	$m = 0$	$m \neq 0$
$e = 0$	$v = (-1)^s\, 0$ (signed zero)	$v = (-1)^s\, m 2^{1-1023}$ (subnormals)
$e = 2^{11} - 1$	$v = (-1)^s\, \infty$	$v = (-1)^s$ NaN (not-a-number)

where $e = 2^{11} - 1 = 11111111111_2 = 2047$. The largest and smallest number that is not infinity is thus

$$e = 2^{11} - 2 = 2046, \tag{B.6}$$

$$m = \sum_{j=1}^{52} 2^{-j} = 1 - 2^{-52} \simeq 1, \tag{B.7}$$

$$v_{\max} = \pm \left(2 - 2^{-52}\right) 2^{1023} \simeq \pm 2^{1024} \simeq \pm 10^{308}. \tag{B.8}$$

The density of numbers varies in such a way that when $e - 1023 = 52$, then

$$v = (-1)^s \left(1 + \sum_{j=1}^{52} m_j 2^{-j}\right) 2^{52} \tag{B.9}$$

$$= \pm \left(2^{52} + \sum_{j=1}^{52} m_j 2^{-j} 2^{52}\right) \tag{B.10}$$

$$= \pm \left(2^{52} + \sum_{j=1}^{52} m_j 2^{52-j}\right) \tag{B.11}$$

$$\overset{k=52-j}{=} \pm \left(2^{52} + \sum_{k=51}^{0} m_{52-k} 2^k\right), \tag{B.12}$$

which are all integers in the range $2^{52} \leq |v| < 2^{53}$. When $e - 1023 = 53$, then the same calculation gives

$$v \overset{k=53-j}{=} \pm \left(2^{53} + \sum_{k=52}^{1} m_{53-k} 2^k\right), \tag{B.13}$$

which are every second integer in the range $2^{53} \leq |v| < 2^{54}$, and so on for larger values of e. When $e - 1023 = 51$, the same calculation gives,

$$v \overset{k=51-j}{=} \pm \left(2^{51} + \sum_{k=50}^{-1} m_{51-k} 2^k\right), \tag{B.14}$$

which is a distance between numbers of $1/2$ in the range $2^{51} \leq |v| < 2^{52}$, and so on for smaller values of e. Thus we may conclude that the distance between numbers in the interval $2^n \leq |v| < 2^{n+1}$ is 2^{n-52}, for $-1022 = 1 - 1023 \leq n < 2046 - 1023 = 1023$. For subnormals, the distance between numbers is

$$v = (-1)^s \left(\sum_{j=1}^{52} m_j 2^{-j}\right) 2^{-1022} \tag{B.15}$$

$$= \pm \left(\sum_{j=1}^{52} m_j 2^{-j} 2^{-1022}\right) \tag{B.16}$$

$$= \pm \left(\sum_{j=1}^{52} m_j 2^{-j-1022}\right) \tag{B.17}$$

$$\overset{k=-j-1022}{=} \pm \left(\sum_{j=-1023}^{-1074} m_{-k-1022} 2^k\right), \tag{B.18}$$

which gives a distance between numbers of $2^{-1074} \simeq 10^{-323}$ in the range $0 < |v| < 2^{-1022} \simeq 10^{-308}$.

Appendix C

Commonly Used Character Sets

Letters, digits, symbols, and space are the core of how we store data, write programs, and communicate with computers and each other. These symbols are in short called characters and represent a mapping between numbers, also known as codes, and a pictorial representation of the character. E.g., the ASCII code for the letter 'A' is 65. These mappings are for short called character sets, and due to differences in natural languages and symbols used across the globe, many different character sets are in use. E.g., the English alphabet contains the letters 'a' to 'z'. These letters are common to many other European languages which in addition use even more symbols and accents. For example, Danish has further the letters 'æ', 'ø', and 'å'. Many non-European languages have completely different symbols, where the Chinese character set is probably the most extreme, and some definitions contain 106,230 different characters, albeit only 2,600 are included in the official Chinese language test at the highest level.

Presently, the most common character set used is Unicode Transformation Format (UTF), whose most popular encoding schemes are 8-bit (UTF-8) and 16-bit (UTF-16). Many other character sets exist, and many of the later build on the American Standard Code for Information Interchange (ASCII). The ISO-8859 codes were an intermediate set of character sets that are still in use, but which is greatly inferior to UTF. Here we will briefly give an overview of ASCII, ISO-8859-1 (Latin1), and UTF.

C.1 ASCII

The *American Standard Code for Information Interchange* (*ASCII*) [8], is a 7 bit code tuned for the letters of the English language, numbers, punctuation symbols, control codes and space, see Tables C.1 and C.2. The first 32 codes are reserved for

x0+0x	00	10	20	30	40	50	60	70
00	NUL	DLE	SP	0	@	P	‘	p
01	SOH	DC1	!	1	A	Q	a	q
02	STX	DC2	"	2	B	R	b	r
03	ETX	DC3	#	3	C	S	c	s
04	EOT	DC4	$	4	D	T	d	t
05	ENQ	NAK	%	5	E	U	e	u
06	ACK	SYN	&	6	F	V	f	v
07	BEL	ETB	’	7	G	W	g	w
08	BS	CAN	(	8	H	X	h	x
09	HT	EM	)	9	I	Y	i	y
0A	LF	SUB	*	:	J	Z	j	z
0B	VT	ESC	+	;	K	[	k	{
0C	FF	FS	,	<	L	\	l	|
0D	CR	GS	–	=	M	]	m	}
0E	SO	RS	.	>	N	^	n	~
0F	SI	US	/	?	O	_	o	DEL

Table C.1 ASCII

non-printable control characters to control printers and similar devices or to provide meta-information. The meaning of each control character is not universally agreed upon.

The code order is known as *ASCIIbetical order*, and it is sometimes used to perform arithmetic on codes, e.g., an uppercase letter with code *c* may be converted to lower case by adding 32 to its code. The ASCIIbetical order also has a consequence for sorting, i.e., when sorting characters according to their ASCII code, ’A’ comes before ’a’, which comes before the symbol ’{’.

C.2 ISO/IEC 8859

The ISO/IEC 8859 report `http://www.iso.org/iso/catalogue_detail?csnumber=28245` defines 10 sets of codes specifying up to 191 codes and graphics characters using 8 bits. Set 1, also known as ISO/IEC 8859-1, Latin alphabet No. 1, or *Latin1*, covers many European languages and is designed to be compatible with ASCII, such that code for the printable characters in ASCII is the same in ISO 8859-1. Table C.3 shows the characters above 7e. Codes 00-1f and 7f-9f are undefined in ISO 8859-1.

Code	Description
NUL	Null
SOH	Start of heading
STX	Start of text
ETX	End of text
EOT	End of transmission
ENQ	Enquiry
ACK	Acknowledge
BEL	Bell
BS	Backspace
HT	Horizontal tabulation
LF	Line feed
VT	Vertical tabulation
FF	Form feed
CR	Carriage return
SO	Shift out
SI	Shift in
DLE	Data link escape
DC1	Device control one
DC2	Device control two
DC3	Device control three
DC4	Device control four
NAK	Negative acknowledge
SYN	Synchronous idle
ETB	End of transmission block
CAN	Cancel
EM	End of medium
SUB	Substitute
ESC	Escape
FS	File separator
GS	Group separator
RS	Record separator
US	Unit separator
SP	Space
DEL	Delete

Table C.2 ASCII symbols.

C.3 Unicode

Unicode is a character standard defined by the Unicode Consortium, `http://unicode.org`, as the *Unicode Standard*. Unicode allows for 1,114,112 different codes. Each code is called a *code point* which represents an abstract character. Code points are divided into 17 planes, each with $2^{16} = 65,536$ code points. Planes are further subdivided into named *blocks*. The first plane is called the *Basic Multilingual plane* and its block of the first 128 code points is called the *Basic Latin block* and is identical to ASCII, see Table C.1, and code points 128-255 are called the *Latin-1 Supplement block*, and are identical to the upper range of ISO 8859-1, see Table C.3. Each code-point has a number of attributes such as the *Unicode general category*. Presently more than 128,000 code points are defined as covering 135 modern and historical writing systems, and obtained

x0+0x	80	90	A0	B0	C0	D0	E0	F0
00			NBSP	°	À	Ð	à	ð
01			¡	±	Á	Ñ	á	ñ
02			¢	²	Â	Ò	â	ò
03			£	³	Ã	Ó	ã	ó
04			¤	´	Ä	Ô	ä	ô
05			¥	µ	Å	Õ	å	õ
06			¦	¶	Æ	Ö	æ	ö
07			§	·	Ç	×	ç	÷
08			¨	¸	È	Ø	è	ø
09			©	¹	É	Ù	é	ù
0a			ª	º	Ê	Ú	ê	ú
0b			«	»	Ë	Û	ë	û
0c			¬	¼	Ì	Ü	ì	ü
0d			SHY	½	Í	Ý	í	ý
0e			®	¾	Î	Þ	î	þ
0f			¯	¿	Ï	ß	ï	ÿ

Table C.3 ISO-8859-1 (latin1) non-ASCII part. Note that the codes 7f – 9f are undefined.

Code	Description
NBSP	Non-breakable space
SHY	Soft hypen

Table C.4 ISO-8859-1 special symbols.

at `http://www.unicode.org/Public/UNIDATA/UnicodeData.txt`, which includes the code point, name, and general category.

A Unicode code point is an abstraction from the encoding and the graphical representation of a character. A code point is written as "U+" followed by its hexadecimal number, and for the Basic Multilingual plane, 4 digits are used, e.g., the code point with the unique name LATIN CAPITAL LETTER A has the Unicode code point "U+0041", and is in this text visualized as 'A'. More digits are used for code points of the remaining planes.

The general category is used to specify valid characters that do not necessarily have a visual representation but possibly transform text. Some categories and their letters in the first 256 code points are shown in Table C.5.

To store and retrieve code points, they must be encoded and decoded. A common encoding is *UTF-8*, which encodes code points as 1 to 4 bytes, and which is backward-compatible with ASCII and ISO 8859-1. Hence, in all 3 coding systems, the character with code 65 represents the character 'A'. Another popular encoding scheme is *UTF-16*, which encodes characters as 2 or 4 bytes, but which is not backward-compatible with ASCII or ISO 8859-1. UTF-16 is used internally in many compilers, interpreters, and operating systems.

General category	Code points	Name
Lu	U+0041–U+005A, U+00C0–U+00D6, U+00D8–U+00DE	Upper case letter
Ll	U+0061–U+007A, U+00B5, U+00DF–U+00F6, U+00F8–U+00FF	Lower case letter
Lt	None	Digraphic letter, with first part uppercase
Lm	None	Modifier letter
Lo	U+00AA, U+00BA	Gender ordinal indicator
Nl	None	Letterlike numeric character
Pc	U+005F	Low line
Mn	None	Nonspacing combining mark
Mc	None	Spacing combining mark
Cf	U+00AD	Soft Hyphen

Table C.5 Some general categories for the first 256 code points.

Appendix D

Common Language Infrastructure

The *Common Language Infrastructure* (*CLI*), not to be confused with *Command Line Interface* with the same acronym, is a technical standard developed by Microsoft [4, 3]. The standard specifies a language, its format, and a runtime environment that can execute the code. The main feature is that it provides a common interface between many languages and many platforms, such that programs can collaborate in a language-agnostic manner and can be executed on different platforms without having to be recompiled. Main features of the standard are:

Common Type System (CTS) which defines a common set of types that can be used across different languages as if it were their own.

Metadata which defines a common method for referencing programming structures such as values and functions in a language-independent manner.

Common Intermediate Language (CIL) which is a platform-independent, stack-based, object-oriented assembly language that can be executed by the Virtual Execution System.

Virtual Execution System (VES) which is a platform dependent, virtual machine, which combines the above into code that can be executed at runtime. Microsoft's implementation of VES is called *Common Language Runtime* (*CLR*) and uses *just-in-time* compilation. In this book, we have been using the `mono` command.

The process of running an F# program is shown in Figure D.1. First the F# code is compiled or interpreted to CIL. This code possibly combined with other CIL code is then converted to a machine-readable code, and the result is then executed on the platform.

CLI defines a *module* as a single file containing executable code by VES. Hence, CLI's notion of a module is somewhat related to F#'s notion of module, but the

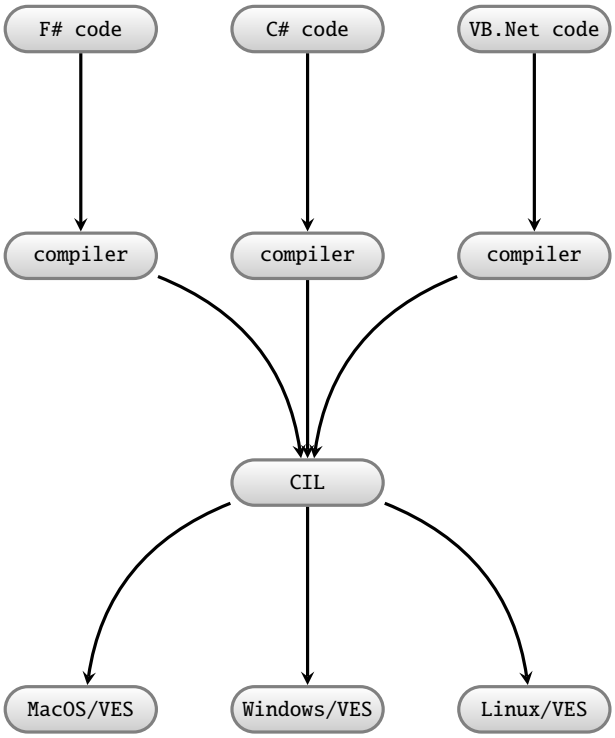


Fig. D.1 The relation between some .NET/Mono languages with the Common intermediate language (CIL), and the Virtual execution systems (VES) on some operating system (`mono`).

two should not be confused. A collection of modules, a *manifest*, and possibly other resources, which jointly define a complete program is called an *assembly*. The manifest is the description of which files are included in the assembly together with its version, name, security information, and other bookkeeping information.

References

1. Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345—-363, 1936.
2. Ole-Johan Dahl and Kristen Nygaard. SIMULA a language for programming and description of discrete event systems. introduction and user's manual. Technical report, Norwegian Computing Center, 1967.
3. European Computer Manufacturers Association (ECMA). Standard ecma-335, common language infrastructure (cli). `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.
4. International Organization for Standardization. Iso/iec 23271:2012, common language infrastructure (cli). `https://www.iso.org/standard/58046.html`.
5. Object Management Group. Uml version 2. `https://www.omg.org/spec/UML/`.
6. Programming Research Group. Specifications for the ibm mathematical formula translating system, fortran. Technical report, Applied Science Division, International Business Machines Corporation, 1954.
7. John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.
8. X3: ASA Sectional Committee on Computers and Information Processing. American standard code for information interchange. Technical Report ASA X3.4-1963, American Standards Association (ASA), 1963. `http://worldpowersystems.com/projects/codes/X3.4-1963/`.
9. George Pólya. *How to solve it*. Princeton University Press, 1945.
10. Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1936.

Index