Learning to Program with F#

Jon Sporring

Department of Computer Science,
University of Copenhagen

2022-07-27 15:21:01+02:00

Contents

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 Mono version 6.0.0.327.

Jon Sporring
Professor, Ph.d.
Department of Computer Science,
University of Copenhagen
2022-07-27

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

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

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 nor programming languages but uses mathematics to prove 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 in order for you to be able to make programs. You do not need to be a master in F# nor to know every corner of the language, and you will expand your knowledge as you expose yourself to solving problems in the language, but you must invest an initial amount of time and energy in order to learn the basics of the language. This book aims at getting you started quickly, which is why we intentionally teach just a small subset of F#. On the internet and through

other works you will be able to learn much more.

3. Practice: In order 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 a number of 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 [**?**] 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 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 a number of implementations which vary in how much code they require, to which degree they rely on external libraries, which programming style they are best suited for, what machine resources they require, and 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 require a 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 [**?**]. The first language emphasizing functional programming was Lisp [**?**].

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 recipe for bread might tell you to first mix yeast and water, then add flour, etc. In imperative programming what should be done are 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 [**?**]. As a historical note, the first major language was FORTRAN [**?**] 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 the 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 [**?**].

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 as, 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 a number of advantages:

Interactive and compile mode: F# has an interactive and a compile mode of oper-

ation. In interactive mode you can write code that is executed immediately in a manner similar to working with a calculator, while in compile mode you combine many lines of code possibly in many files into a single application, which is easier to distribute to 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 byte-code 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 **??**–**??**, functional

programming mainly covered in **??–??**, object-oriented programming in **????**, and event-driven programming in **??**. A number of general topics are given in the appendix for reference. For further reading please consult `http://fsharp.org`.

2 Executing F# Code

2.1 Source Code

F# is a functional first programming language, meaning that it has strong support for functional programming, but also supports imperative and object-oriented programming.

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. Interactive mode is well suited for small experiments or back-of-an-envelope calculations, but not for programming in general. In compile mode, the user writes a complete program, which is translated or compiled using the F# compiler into a compiled file. The compiled file can be run or executed as a stand-alone program using a virtual machine called `mono`. In both the interactive and compile mode, F# statements are translated into something that can be executed on the computer. A major difference is that in interactive mode, the translation is performed everytime the program is executed, while in compiled mode the translation is performed only once.

Both interactive and compile modes can be accessed via the *console*, see **??** for more information on the console. The interactive system is started by calling `fsharpi` at the command prompt in the console, while compilation is performed with `fsharpc`, and execution of the compiled code is performed using the `mono` command.

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

.fs An *implementation file*, e.g., `myModule.fs`

.fsi A *signature file*, e.g., `myModule.fsi`

.fsx A *script file*, e.g., `gettingStartedStump.fsx`

.fsscript Same as .fsx, e.g., `gettingStartedStump.fsscript`

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

.dll A *library file*, e.g., `myModule.dll`

.exe A stand-alone *executable file*, e.g., `gettingStartedStump.exe`

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

2.2 Executing Programs

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

To illustrate the difference between interactive and compile mode, consider the program in Listing 3.2. The code declares a value `a` to be the decimal value 3.0 and

---

Listing 2.1 gettingStartedStump.fsx:
A simple demonstration script.

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

---

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

An interactive session is obtained by starting the console, typing the `fsharpi` command, typing the lines of the program, and ending the script-fragment with ";;". The dialogue in Listing 3.3 demonstrates the workflow. What the user types has been highlighted by a box.

Listing 2.2: An interactive session.

```
1  $ fsharpi
2
3  F# Interactive for F# 4.1 (Open Source Edition)
4  Freely distributed under the Apache 2.0 Open Source License
5
6  For help type #help;;
7
8  > let a = 3.0
9  - do printfn "%g" a;;
10 3
11
12 val a : float = 3.0
13 val it : unit = ()
14
15 > #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 = 3.0` and presses `enter`, to which the interpreter responds with `-`. This indicates that the line has been received, that the script-fragment is not yet completed, and that it is ready to receive more input. When the user types `do printfn "%g" a;;` followed by `enter`, then by ";;" the interpreter knows that the script-fragment is completed, it interprets the script-fragment, responds with `3` and 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 3.3 F# states that the name `a` has *type* `float` and the value `3.0`. 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 ingrediens 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 3.2 into a file, here called `gettingStartedStump.fsx`. This file can be interpreted directly by `fsharpi` as shown in Listing 3.4.

Listing 2.3: Using the interpreter to execute a script.

```
1  $ fsharpi gettingStartedStump.fsx
2  3
```

Notice that in the file, ";;" is optional. We see that the interpreter executes the code

and prints the result on screen without the extra type information as compared to Listing 3.3.

Finally, the file containing Listing 3.2 may be compiled into an executable file with the program `fsharpc`, and run using the program `mono` from the console. This is demonstrated in Listing 3.5.

Listing 2.4: Compiling and executing a script.

```
1  $ fsharpc gettingStartedStump.fsx
2  F# Compiler for F# 4.1 (Open Source Edition)
3  Freely distributed under the Apache 2.0 Open Source License
4  $ mono gettingStartedStump.exe
5  3
```

The compiler reads `gettingStartedStump.fsx` and makes `gettingStartedStump.exe`, which can be run using `mono`.

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

Command	Time
`fsharpi gettingStartedStump.fsx`	1.88s
`fsharpc gettingStartedStump.fsx`	1.90s
`mono gettingStartedStump.exe`	0.05s

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

Executing programs with the interpreted directly from a file and compiling and executing the program is much preferred for programming complete programs, since the starting state is well defined, and since this better supports *unit testing*, which is a method for debugging programs. Thus, **prefer compiling over interpretation.**

2.3 The Printf Function

A common way to output information to the console is to use one of the family of *printf* commands. These functions are special, since they take a variable number of arguments, and the number is decided by the first argument - the format string. The syntax for the `printf` commands are as follows:

```
Listing 2.5:  printf statement.

1  printf <format-string> {<ident>}
```

The `formatString` is a string (simple or verbatim) with placeholders. The function `printf` prints `formatString` to the console, where all `placeholder` have been replaced by the values of the corresponding arguments formatted as specified. For example, in `printfn "1 2 %d" 3`, the `formatString` is `"1 2 %d"` and the placeholder is `%d`. When executed, `printf` will replace `%d` with the following argument, `3`, and print the result to the console: `1 2 3`. There are specifiers for all the basic types, and more, as elaborated in Table 2.1. The placeholder can be parametrized, e.g., the

Specifier	Type	Comment
%b	bool	formatted as "true" or "false"
%s	string	
%c	char	
%d, %i	basic integer	
%u	basic unsigned integers	
%x	basic integer	formatted as unsigned hexadecimal with lower case letters
%X	basic integer	formatted as unsigned hexadecimal with upper case letters
%o	basic integer	formatted as unsigned octal integer
%f, %F,	basic floats	formatted on decimal form
%e, %E,	basic floats	formatted on scientific form. Lower case uses "e" while upper case uses "E" in the formatting.
%g, %G,	basic floats	formatted on the shortest of the corresponding decimal or scientific form.
%A, %O	any value	formatted as a basic type or using the object's `ToString` method

Table 2.1: Some printf placeholder strings.

placeholder string `%8s` will print a right-aligned string which that is eight characters

wide and padded with spaces, as needed. For floating point numbers, `%8f` will print a number that is exactly seven digits and a decimal point, making eight characters in total. Zeros are added after the decimal point, as needed. Alternatively, we may specify the number of digits after the decimal point, such that `%8.1f` will print a floating point number, aligned to the right, with one digit after the decimal point padded with spaces, as needed. The default is for the value to be right justified in the field, but left justification can be specified by the - character. For number types, you can specify their format by `"0"` for padding the number with zeros to the left when right justifying the number; `"+"` to explicitly show a plus sign for positive numbers; `SP` to enforce a space, where there otherwise would be a plus sign for positive numbers. The placeholder parameter may also be given as an argument to `printf` which case the placeholder should use the `*` character instead of an integer.

Examples of placeholder parametrization are shown in Listing 2.6. Not all combinations of flags and identifier types are supported, e.g., strings cannot have the number of integers after the decimal point specified. The placeholder types `"%A"`, `"%a"`, and `"%t"` are special for F#, examples of their use are shown in Listing 2.7. The `%A` is special in that all built-in types, including tuples, lists, and arrays to be discussed in **??**, can be printed using this formatting string, but notice that the formatting performed includes the named literal string. The two formatting strings `%t` and `%a` are options for user-customizing the formatting and will not be discussed further.

Beware, `formatString` is not a `string` but a `Printf.TextWriterFormat`, so to pre-define a `formatString` as, e.g., `let str = "hello %s" in printf str "world"`, will be a type error.

The family of `printf` is shown in Table 2.2. The function `fprintf` prints to a stream, e.g., `stderr` and `stdout`, of type `System.IO.TextWriter`. For the moment it is sufficient to think of both `stderr` and `stdout` to be the console. Streams will be discussed in further detail in **??**. The function `failwithf` is used with exceptions, see **??** for more details. The function has a number of possible return value types, and for testing, the *ignore* function ignores it all, e.g., `ignore (failwithf "%d failed apples" 3)`.

2.4 Reading from the Console

The `printf` and `printfn` functions allow us to write text on the screen. A program often needs to ask a user to input data, e.g., by typing text on a keyboard. Text typed on the keyboard is accessible through the `stdin` stream, and F# provides several library functions for capturing text typed on the keyboard. In the following section, we will briefly discuss the `System.Console.ReadLine` function. For more details and other methods of input see **????**.

Listing 2.6 printfExample.fsx:
Examples of printf and some of its formatting options.

```
1  let pi = 3.1415192
2  let hello = "hello"
3  printf "An integer: %d\n" (int pi)
4  printf "A float %f on decimal form and on %e scientific form\n"
      pi pi
5  printf "A char '%c' and a string \"%s\"\n" hello.[0] hello
6  printf "Float using width 8 and 1 number after the decimal:\n"
7  printf "   \"%8.1f\" \"%8.1f\"\n" pi -pi
8  printf "   \"%08.1f\" \"%08.1f\"\n" pi -pi
9  printf "   \"% 8.1f\" \"% 8.1f\"\n" pi -pi
10 printf "   \"%-8.1f\" \"%-8.1f\"\n" pi -pi
11 printf "   \"%+8.1f\" \"%+8.1f\"\n" pi -pi
12 printf "   \"%8s\"\n\"%-8s\"\n" "hello" "hello"
```

```
1  $ fsharpc --nologo printfExample.fsx && mono
      printfExample.exe
2  An integer: 3
3  A float 3.141519 on decimal form and on 3.141519e+000
      scientific form
4  A char 'h' and a string "hello"
5  Float using width 8 and 1 number after the decimal:
6     "     3.1" "    -3.1"
7     "000003.1" "-00003.1"
8     "     3.1" "    -3.1"
9     "3.1     " "-3.1    "
10    "    +3.1" "    -3.1"
11    "   hello"
12 "hello   "
```

The function `System.Console.ReadLine` takes a unit value as an argument and re-
turns the string the user typed. The program will not advance until the user presses
newline. An example of a program that multiplies two floating point numbers sup-
plied by a user is given in Listing 2.8, and an example dialogue is shown in Listing 2.9.
Note that the string is immediately cast to floats such that we can multiply the input
using the float multiplication operator. This also implies that if the user inputs a
non-number, then `mono` will halt with an error message.

> **Listing 2.7 printfExampleAdvance.fsx:**
> **Custom format functions may be used to specialise output.**
>
> ```
> 1  let noArgument writer = printf "I will not print anything"
> 2  let customFormatter writer arg = printf "Custom formatter got:
>      \"%A\"" arg
> 3  printf "Print examples: %A, %A, %A\n" 3.0m 3uy "a string"
> 4  printf "Print function with no arguments: %t\n" noArgument
> 5  printf "Print function with 1 argument: %a\n" customFormatter 3.0
> ```
> --
> ```
> 1  $ fsharpc --nologo printfExampleAdvance.fsx && mono
>      printfExampleAdvance.exe
> 2  Print examples: 3.0M, 3uy, "a string"
> 3  Print function with no arguments: I will not print anything
> 4  Print function with 1 argument: Custom formatter got: "3.0"
> ```

Function	Example	Description
`printf`	`printf "%d apples" 3`	Prints to the console, i.e., `stdout`
`printfn`		As `printf` and adds a newline.
`fprintf`	`fprintf stream "%d apples" 3`	Prints to a stream, e.g., `stderr` and `stdout`, which would be the same as `printf` and `eprintf`.
`fprintfn`		As `fprintf` but with added newline.
`eprintf`	`eprintf "%d apples" 3`	Prints to `stderr`
`eprintfn`		As `eprintf` but with added newline.
`sprintf`	`sprintf "%d apples" 3`	Return printed string
`failwithf`	`failwithf "%d failed apples" 3`	Prints to a string and used for raising an exception.

Table 2.2: The family of printf functions.

Listing 2.8 userDialoguePrintf.fsx:
Interacting with a user using `ReadLine`**.**

```
1  printfn "To perform the multiplication of a and b"
2  printf "Enter a: "
3  let a = float (System.Console.ReadLine ())
4  printf "Enter b: "
5  let b = float (System.Console.ReadLine ())
6  printfn "a * b = %A" (a * b)
```

Listing 2.9: An example dialogue of running Listing 2.8. The user
typed "3.5" and "7.4".

```
1  $ fsharpc --nologo userDialoguePrintf.fsx && mono
     userDialoguePrintf.exe
2  To perform the multiplication of a and b
3  Enter a: 3.5
4  Enter b: 7.4
5  a * b = 25.9
```

3 Solving problems by writing a program

Programming is the art of solving problems by writing a program to be executed by a computer. For example, to solve the following problem,

---

Task 3.1

What is the sum of 357 and 864?

---

we have written the program shown in Listing 3.1. In the box the above, we

---

Listing 3.1 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
```

```
1  $ fsharpc --nologo quickStartSum.fsx && mono
     quickStartSum.exe
2  1221
```

---

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

To solve the problem, we made program consisting of several lines, where each line was an *expressions*. The first expression, `let a = 357`, in line 1 used the *let keyword*

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

3.1 Executing F# programs on a computer

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. Interactive mode is well suited for small experiments or back-of-an-envelope calculations, but not for programming in general. In compile mode, the user writes a complete program, which is translated or compiled using the F# compiler into a compiled file. The compiled file can be run or executed as a stand-alone program using a virtual machine called `mono`. In both the interactive and compile mode, F# statements are translated into something that can be executed on the computer. A major difference is that in interactive mode, the translation is performed everytime the program is executed, while in compiled mode the translation is performed only once.

Both interactive and compile modes can be accessed via the *console*, see **??** for more information on the console. The interactive system is started by calling `fsharpi` at the command prompt in the console, while compilation is performed with `fsharpc`, and execution of the compiled code is performed using the `mono` command.

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

`.fs` An *implementation file*, e.g., `myModule.fs`

`.fsi` A *signature file*, e.g., `myModule.fsi`

`.fsx` A *script file*, e.g., `gettingStartedStump.fsx`

`.fsscript` Same as `.fsx`, e.g., `gettingStartedStump.fsscript`

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

`.dll` A *library file*, e.g., `myModule.dll`

`.exe` A stand-alone *executable file*, e.g., `gettingStartedStump.exe`

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

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

To illustrate the difference between interactive and compile mode, consider the program in Listing 3.2. The code declares a value `a` to be the decimal value 3.0 and

---

Listing 3.2 gettingStartedStump.fsx:
A simple demonstration script.

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

---

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

An interactive session is obtained by starting the console, typing the `fsharpi` command, typing the lines of the program, and ending the script-fragment with "`;;`". The dialogue in Listing 3.3 demonstrates the workflow. What the user types has been highlighted by a box.

```
   Listing 3.3: An interactive session.
1  $ fsharpi
2
3  F# Interactive for F# 4.1 (Open Source Edition)
4  Freely distributed under the Apache 2.0 Open Source License
5
6  For help type #help;;
7
8  > let a = 3.0
9  - do printfn "%g" a;;
10 3
11
12 val a : float = 3.0
13 val it : unit = ()
14
15 > #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 = 3.0` and presses `enter`, to which the interpreter responds with `-`. This indicates that the line has been received, that the script-fragment is not yet completed, and that it is ready to receive more input. When the user types `do printfn "%g" a;;` followed by `enter`, then by "`;;`" the interpreter knows that the script-fragment is completed, it interprets the script-fragment, responds with `3` and 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 3.3 F# states that the name `a` has *type* `float` and the value `3.0`. 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 ingrediens 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 3.2 into a file, here called `gettingStartedStump.fsx`. This file can be interpreted directly by `fsharpi` as shown in Listing 3.4.

Listing 3.4: Using the interpreter to execute a script.

```
1  $ fsharpi gettingStartedStump.fsx
2  3
```

Notice that in the file, "`;;`" is optional. We see that the interpreter executes the code and prints the result on screen without the extra type information as compared to Listing 3.3.

Finally, the file containing Listing 3.2 may be compiled into an executable file with the program `fsharpc`, and run using the program `mono` from the console. This is demonstrated in Listing 3.5.

Listing 3.5: Compiling and executing a script.

```
1  $ fsharpc gettingStartedStump.fsx
2  F# Compiler for F# 4.1 (Open Source Edition)
3  Freely distributed under the Apache 2.0 Open Source License
4  $ mono gettingStartedStump.exe
5  3
```

The compiler reads `gettingStartedStump.fsx` and makes `gettingStartedStump.exe`, which can be run using `mono`.

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

Command	Time
`fsharpi gettingStartedStump.fsx`	1.88s
`fsharpc gettingStartedStump.fsx`	1.90s
`mono gettingStartedStump.exe`	0.05s

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

Executing programs with the interpreted directly from a file and compiling and executing the program is much preferred for programming complete programs, since the starting state is well defined, and since this better supports *unit testing*, which is a method for debugging programs. Thus, **prefer compiling over interpretation.**

3.2 Values have types and types reduce risk of programming errors

Types are a central concept in F#. In the script 3.1 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 **??**. The interactive session displays the type using

Listing 3.6: Inferred types are given as part of the response from the interpreter.

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

the *val* keyword followed by the name used in the binding, its type, and its value. Since the value is also responded, 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".

The following problem,

Task 3.2

What is the sum of 357.6 and 863.4?

uses *decimal point* numbers instead of integers. These numbers are called *floating point numbers*, which is a subset of the set of *reals*. 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 $\mathbb{N} \subset \mathbb{Z} \subset \mathbb{R}$. Since computers have finite memory, working with such infinite sets may be possible but 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 $[-2,147,483,648 \ldots 2,147,483,647]$. The most commonly used type to represent real-like numbers is called `float` and is a clever selection of 2^{64} rational numbers, which have proven useful in practice. Since the internal representation of each type on the computer differs substantially, is quite different to integer numbers used previously. Likewise, algorithms used to perform arithmetic are quite different from integers. Now the program would look like **??**. On the surface, this could

---

Listing 3.7 quickStartSumFloat.fsx:
Floating point types and arithmetic.

```
1  let a = 357.6
2  let b = 863.4
3  let c = a + b
4  do printfn "%A" c
```
- -
```
1  $ fsharpc --nologo quickStartSumFloat.fsx && mono
     quickStartSumFloat.exe
2  1221.0
```

---

appear as a negligible change, but the set of integers and the set of real numbers (floats) require quite different representations in order to be effective on a computer, and as a consequence, the implementation of their operations, such as addition, are very different. Thus, although the response is an integer, it has type `float` which is indicated by `1221.0` and which is not the same as `1221`. F# is very picky about types, and generally does not allow types to be mixed, as demonstrated in the interactive session in **??**. We see that binding a name to a number without a decimal point is inferred to be an integer, while when binding to a number with a decimal point the type is inferred to be a float, and that our attempt of adding an integer and floating point value gives an error. The *error message* contains much information. First, it states that the error is in `stdin(4,13)`, which means that the error was found on standard-input at line 4 and column 13. Since the program was executed using `fsharpi quickStartSumFloat.fsx`, here standard input means the file `quickStartSumFloat.fsx` shown in **??**. The corresponding line and column are also shown in **??**. After the file, line, and column number, F# informs us of the error

Listing 3.8: Mixing types is often not allowed.

```
1   > let a = 357;;
2   val a : int = 357
3
4   > let b = 863.4;;
5   val b : float = 863.4
6
7   > let c = a + b;;
8
9     let c = a + b;;
10    ------------^
11
12  stdin(4,13): error FS0001: The type 'float' does not match
      the type 'int'
```

number and a description of the error. Error numbers are an underdeveloped feature in Mono and should be ignored. However, the verbal description often contains useful information for *debugging*. In the example we are informed that there is a type mismatch in the expression, i.e., since a is an integer, F# expected b to be one too. Debugging is the process of solving errors in programs, and here we can solve the error by either making a into a float or b into an int. The right solution depends on the application.

3.3 Controling the flow of execution

- conditions

- recursion

- patterns

- for and while loops

3.4 Conditional Expressions

Programs often contain code which should only be executed under certain conditions. This can be expressed with `if`-expressions, whose syntax is as follows.

> **Listing 3.9:** Conditional expressions.
>
> ```
> 1  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 evaluate. If all conditions are false then the `else`-branch is evaluated. If no `else` expression is present, then "()" will be returned. See **??** for a simple example. The

> **Listing 3.10 condition.fsx:**
> **Conditions evaluate their branches depending on the value of the condition.**
>
> ```
> 1  if true then printfn "hi" else printfn "bye"
> 2  if false then printfn "hi" else printfn "bye"
> ```
> -
> ```
> 1  $ fsharpc --nologo condition.fsx && mono condition.exe
> 2  hi
> 3  bye
> ```

lightweight syntax allows for newlines entered everywhere, but indentation must be used to express scope.

To demonstrate conditional expressions, let us write a program which writes the sentence "I have n apple(s)", where the plural 's' is added appropriately for various n's. This is done in **??**, using the lightweight syntax. The sentence structure and its variants give rise to a more compact solution, since the language to be returned to the user is a variant of "I have/owe no/number apple(s)", i.e., certain conditions determine whether the sentence should use "have" and "owe" and so forth. So, we could instead make decisions on each of these sentence parts, and then built the final sentence from its parts. This is accomplished in the following example: While arguably shorter, this solution is also denser, and most likely more difficult to debug and maintain.

Listing 3.11 conditionalLightweight.fsx:
Using conditional expression to generate different strings.

```fsharp
let applesIHave n =
  if n < -1 then
    "I owe " + (string -n) + " apples"
  elif n < 0 then
    "I owe " + (string -n) + " apple"
  elif n < 1 then
    "I have no apples"
  elif n < 2 then
    "I have 1 apple"
  else
    "I have " + (string n) + " apples"

printfn "%A" (applesIHave -3)
printfn "%A" (applesIHave -1)
printfn "%A" (applesIHave 0)
printfn "%A" (applesIHave 1)
printfn "%A" (applesIHave 2)
printfn "%A" (applesIHave 10)
```

```
$ fsharpc --nologo conditionalLightWeight.fsx && mono
    conditionalLightWeight.exe
"I owe 3 apples"
"I owe 1 apple"
"I have no apples"
"I have 1 apple"
"I have 2 apples"
"I have 10 apples"
```

Listing 3.12 conditionalLightweightAlt.fsx:
Using sentence parts to construct the final sentence.

```
1  let applesIHave n =
2    let haveOrOwe = if n < 0 then "owe" else "have"
3    let pluralS = if (n = 0) || (abs n) > 1 then "s" else ""
4    let number = if n = 0 then "no" else (string (abs n))
5
6    "I " + haveOrOwe + " " + number + " apple" + pluralS
7
8  printfn "%A" (applesIHave -3)
9  printfn "%A" (applesIHave -1)
10 printfn "%A" (applesIHave 0)
11 printfn "%A" (applesIHave 1)
12 printfn "%A" (applesIHave 2)
13 printfn "%A" (applesIHave 10)
```

```
1  $ fsharpc --nologo conditionalLightWeightAlt.fsx && mono
     conditionalLightWeightAlt.exe
2  "I owe 3 apples"
3  "I owe 1 apple"
4  "I have no apples"
5  "I have 1 apple"
6  "I have 2 apples"
7  "I have 10 apples"
```

4 Recursion

Recursion is a central concept in F# and is used to control flow in loops without the `for` and `while` constructions. **??** illustrates the concept of an infinite loop with recursion.
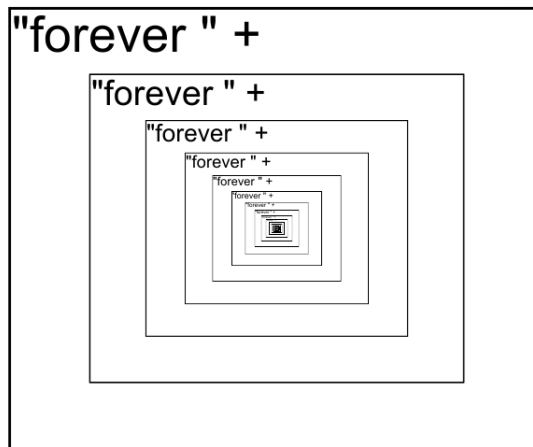
Figure 4.1: An infinitely long string of "forever forever forever...", conceptually calculated by `let rec` `forever ()` = `"forever "` + `(forever ())`.

4.1 Recursive Functions

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

Listing 4.1: Syntax for defining one or more mutually dependent recursive functions.

```
1  let rec <ident> = <expr> {and <ident> = <expr>} [in] <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, then they must be defined jointly using the `and` keyword.

An example of a recursive function that counts from 1 to 10 similarly to **??** is given in **??**. Here the `prt` function calls itself repeatedly, such that the first call is `prt 1 10`,

Listing 4.2 countRecursive.fsx: Counting to 10 using recursion.

```
1  let rec prt a b =
2    if a > b then
3      printf "\n"
4    else
5      printf "%d " a
6      prt (a + 1) b
7
8  prt 1 10
```

```
1  $ fsharpc --nologo countRecursive.fsx && mono
     countRecursive.exe
2  1 2 3 4 5 6 7 8 9 10
```

which calls `prt 2 10`, and so on until the last call `prt 11 10`. Each time `prt` is called, new bindings named `a` and `b` are made to new values. This is illustrated in **??**. The old values are no longer accessible, as indicated by subscripts in the figure. E.g., in prt_3, the scope has access to a_3 but not a_2 and a_1. Thus, in this program, the process is similar to a `for` loop, where the counter is `a`, and in each loop its value is reduced.

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

Listing 4.3: Recursive functions consist of a stopping criterium, a stopping expression, and a recursive step.

```
1  let rec f a =
2    if <stopping condition>
3    then <stopping step>
4    else <recursion step>
```

The *match – with* are also very common conditional structures. In **??**, a > b is the *stopping condition*, `printfn "\n"` is the *stopping step*, and `printfn "\%d " a`;
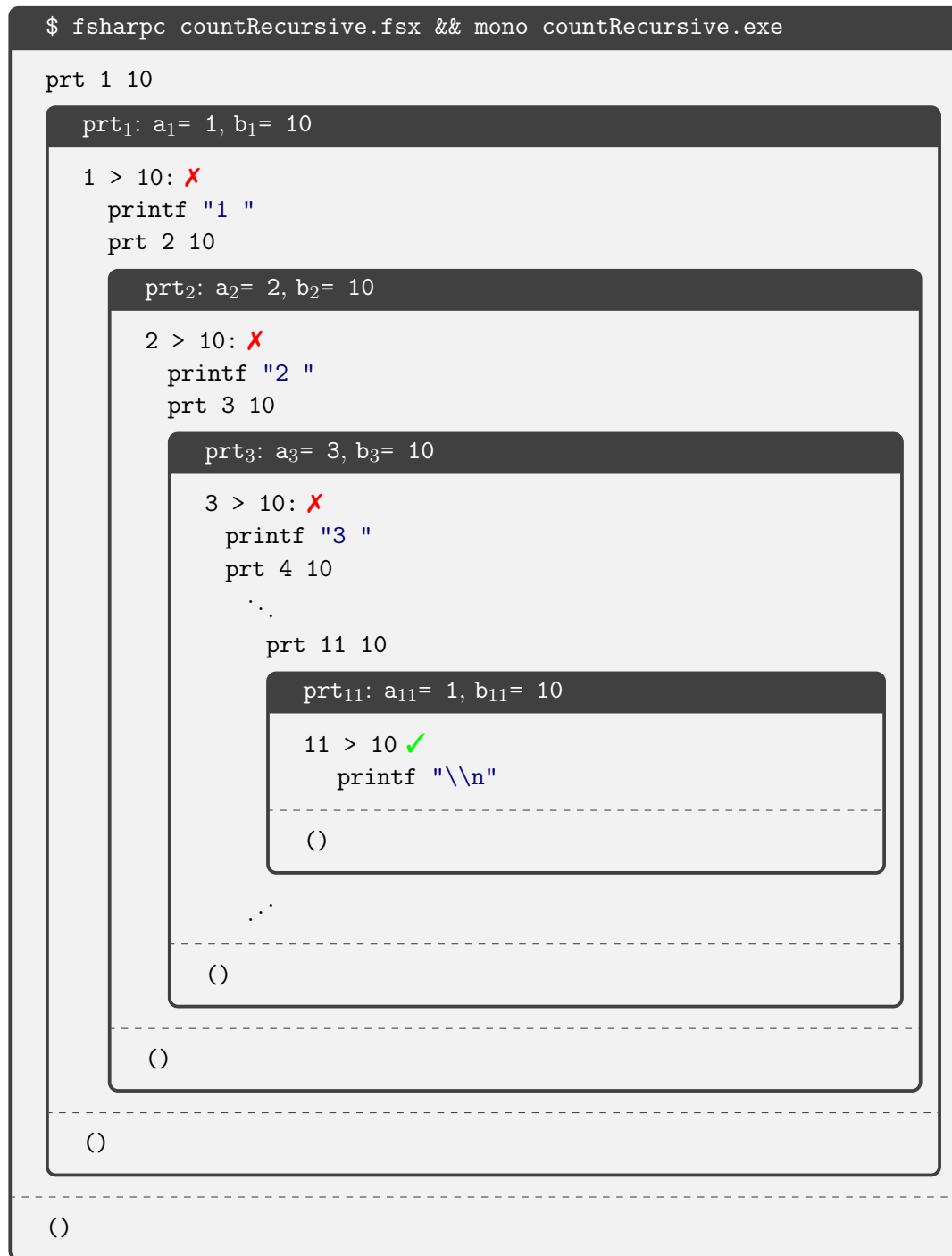
`prt (a + 1) b` is the *recursion step*.

```
$ fsharpc countRecursive.fsx && mono countRecursive.exe
```

```
prt 1 10
```

prt_1: a_1= 1, b_1= 10

```
1 > 10: ✗
  printf "1 "
  prt 2 10
```

prt_2: a_2= 2, b_2= 10

```
2 > 10: ✗
  printf "2 "
  prt 3 10
```

prt_3: a_3= 3, b_3= 10

```
3 > 10: ✗
  printf "3 "
  prt 4 10
```
⋱
```
    prt 11 10
```

prt_{11}: a_{11}= 1, b_{11}= 10

```
11 > 10 ✓
  printf "\\n"
```
- -
```
()
```

⋰

- -
```
()
```

- -
```
()
```

- -
```
()
```

Figure 4.2: Illustration of the recursion used to write the sequence "1 2 3 ... 10" in line **??** in **??**. Each frame corresponds to a call to prt, where new values overshadow old ones. All calls return unit.

5 Pattern Matching

Pattern matching is used to transform values and variables into a syntactical structure. The simplest example is value-bindings. The *let*-keyword was introduced in ??, its extension with pattern matching is given as,

Listing 5.1: Syntax for let-expressions with pattern matching.

```
[[<Literal>]]
let [mutable] <pat> [: <returnType>] = <bodyExpr> [in
    <expr>]
```

A typical use of this is to extract elements of tuples, as demonstrated in ??. Here

Listing 5.2 letPattern.fsx:
Patterns in let expressions may be used to extract elements of tuples.

```
let a = (3,4)
let (x,y) = a
let (alsoX,_) = a
printfn "%A: %d %d %d" a x y alsoX
```

```
$ fsharpc --nologo letPattern.fsx && mono letPattern.exe
(3, 4): 3 4 3
```

we extract the elements of a pair twice. First by binding to x and y, and second by binding to alsoX while using the wildcard pattern to ignore the second element. Thus, again the wildcard pattern in value-bindings is used to underline a disregarded value.

Another common use of patterns is as an alternative to if − then − else expressions, particularly when parsing input for a function. Consider the example in ??. In the example, a discriminated union and a function are defined. The function converts each case to a supporting statement, using an if-expression. The same can be done with the *match − with* expression and patterns, as demonstrated in ??. Here we used a pattern for the discriminated union cases and a wildcard pattern as default.

```
   Listing 5.3 switch.fsx:
   Using if − then − else to print discriminated unions.

1  type Medal = Gold | Silver | Bronze
2  let statement (m : Medal) : string =
3    if m = Gold then "You won"
4    elif m = Silver then "You almost won"
5    else "Maybe you will win next time"
6
7  let m = Silver
8  printfn "%A : %s" m (statement m)
```
```
1  $ fsharpc --nologo switch.fsx && mono switch.exe
2  Silver : You almost won
```

The lightweight syntax for `match`-expressions is,

```
   Listing 5.5:   Syntax for match-expressions.

1  match <inputExpr> with
2    [| ]<pat> [when <guardExpr>] -> <caseExpr>
3    | <pat> [when <guardExpr>] -> <caseExpr>
4    | <pat> [when <guardExpr>] -> <caseExpr>
5    ...
```

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. In lightweight syntax, 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.

Patterns are also used in a version of *for*-loop expressions, and its lightweight syntax is given as,

```
   Listing 5.6:   Syntax for for-expressions with pattern matching.

1  for <pat> in <sourceExpr> do
2    <bodyExpr>
```

Typically, `<sourceExpr>` is a list or an array. An example is given in **??**.   The

39

Listing 5.4 switchPattern.fsx:
Using match − with to print discriminated unions.

```
1  type Medal = Gold | Silver | Bronze
2  let statement (m : Medal) : string =
3    match m with
4      Gold -> "You won"
5      | Silver -> "You almost won"
6      | _ -> "Maybe you can win next time"
7
8  let m = Silver
9  printfn "%A : %s" m (statement m)
```

```
1  $ fsharpc --nologo switchPattern.fsx && mono
     switchPattern.exe
2  Silver : You almost won
```

Listing 5.7 forPattern.fsx:
Patterns may be used in for-loops.

```
1  for (_,y) in [(1,3); (2,1)] do
2    printfn "%d" y
```

```
1  $ fsharpc --nologo forPattern.fsx && mono forPattern.exe
2  3
3  1
```

wildcard pattern is used to disregard the first element in a pair while iterating over the complete list. It is good practice to **use wildcard patterns to emphasize unused values.**

The final expression involving patterns to be discussed is the *anonymous functions*. Patterns for anonymous functions have the syntax,

Listing 5.8: Syntax for anonymous functions with pattern matching.

```
1  fun <pat> [<pat> ...] -> <bodyExpr>
```

This is an extension of the syntax discussed in **??**. A typical use for patterns in *fun*-expressions is shown in **??**. Here we use an anonymous function expression and bind it to f. The expression has one argument of any type, which it ignores through the

---

Listing 5.9 funPattern.fsx:
Patterns may be used in `fun`-expressions.

```
1  let f = fun _ -> "hello"
2  printfn "%s" (f 3)
```

```
1  $ fsharpc --nologo funPattern.fsx && mono funPattern.exe
2  hello
```

---

wildcard pattern. Some limitations apply to the patterns allowed in `fun`-expressions. The wildcard pattern in `fun`-expressions are often used for *mockup functions*, where the code requires the said function, but its content has yet to be decided. Thus, mockup functions can be used as loose place-holders while experimenting with program design.

Patterns are also used in exceptions to be discussed in **??**, and in conjunction with the `function`-keyword, a keyword we discourage in this book. We will now demonstrate a list of important patterns in F#.

5.1 While and For Loops

Many programming constructs need to be repeated, and F# contains many structures for repetition. A *while*-loop has the following syntax:

Listing 5.10:   While loop.

```
1  while <condition> do <expr> [done]
```

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. Using lightweight syntax, the block following the *do* keyword up to and including the *done* keyword may be replaced by a newline and indentation.

The program in **??** is an example of a while-loop which counts from 1 to 10. The variable `i` is customarily called the counter variable. The counting is done by performing the following computation: In line **??**, the counter variable is first given an initial value of 1. Then execution enters the while-loop and examines the condition. Since $1 <= 10$, the condition is true, and execution enters the body of the loop. The body prints the value of the counter to the screen and increases the counter by 1.

Listing 5.11 countWhile.fsx:
Count to 10 with a counter variable.

```
1  let mutable i = 1 in while i <= 10 do printf "%d " i; i <- i + 1
      done;
2  printf "\n"
```

```
1  $ fsharpc --nologo countWhile.fsx && mono countWhile.exe
2  1 2 3 4 5 6 7 8 9 10
```

Then execution returns to the top of the while-loop. 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 **??**.

In lightweight syntax, this would be as shown in **??**. Notice that although the

Listing 5.12 countWhileLightweight.fsx:
Count to 10 with a counter variable using lightweight syntax.

```
1  let mutable i = 1
2  while i <= 10 do
3    printf "%d " i
4    i <- i + 1
5  printf "\n"
```

```
1  $ fsharpc --nologo countWhileLightweight.fsx && mono
      countWhileLightweight.exe
2  1 2 3 4 5 6 7 8 9 10
```

expression following the condition is preceded with a `do` keyword, and `do <expr>` is a `do`-binding, the keyword `do` is mandatory.

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 5.13:   For loop.

```
1  for <ident> = <firstExpr> to <lastExpr> do <bodyExpr> [done]
```

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>`. As for while-loops, when using lightweight syntax the block following the *do* keyword up to and including the *done* keyword may be replaced by a newline and indentation.

The counting example from **??** using a `for`-loop is shown in **??** As this interactive

Listing 5.14 count.fsx:
Counting from 1 to 10 using a `for`-loop.

```
1   for i = 1 to 10 do printf "%d " i done
2   printfn ""
```

```
1   $ fsharpc --nologo count.fsx && mono count.exe
2   1 2 3 4 5 6 7 8 9 10
```

script demonstrates, the identifier `i` takes all the values between 1 and 10, but in spite of its changing state, it is not mutable. Note also that the return value of the `for` expression is "`()`", like the `printf` functions. The lightweight equivalent is shown in **??**.

Listing 5.15 countLightweight.fsx:
Counting from 1 to 10 using a `for`-loop using the lightweight syntax.

```
1   for i = 1 to 10 do
2     printf "%d " i
3   printfn ""
```

```
1   $ fsharpc --nologo countLightweight.fsx && mono
      countLightweight.exe
2   1 2 3 4 5 6 7 8 9 10
```

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 5.16 countLightweightBackwards.fsx:
Counting from 10 to 1 using a `for`-`downto`-loop using the lightweight syntax.

```
1  for i = 10 downto 1 do
2    printf "%d " i
3  printfn ""
```

```
1  $ fsharpc --nologo countLightweightBackwards.fsx && mono
     countLightweightBackwards.exe
2  10 9 8 7 6 5 4 3 2 1
```

5.2 The scope of programming constructs

F# is a functional first programming language, and one implication of this is that names have a *lexical scope*. A scope is the lines in a program where a binding is valid, and lexical scope means that to find the value of a name, F# looks for the value in the above lines. Furthermore, at the outermost level, rebinding is not allowed. If attempted, then F# will return an error as shown in **??**. However, if the same

Listing 5.17 quickStartRebindError.fsx:
A name cannot be rebound.

```
1  let a = 357
2  let a = 864
```

```
1  $ fsharpc --nologo -a quickStartRebindError.fsx
2
3  quickStartRebindError.fsx(2,5): error FS0037: Duplicate
     definition of value 'a'
```

code is executed in an interactive session, then rebinding does not cause an error, as shown in **??**. The difference is that the *";;" lexeme* is used to specify the end of a *script-fragment*. A lexeme is a letter or word, which F# considers as an atomic unit. Script-fragments may be defined both in scripts and in interactive mode, and rebinding is not allowed at the outermost level in script-fragments. Even with the ";;" lexeme, rebinding is not allowed in compile-mode. In general, **avoid rebinding of names.**

In F#, *functions* are also values, and we may define a function `sum` as part of the solution to the above program, as shown in **??**. Functions are useful to *encapsulate*

Listing 5.18: Names may be reused when separated by the lexeme ";;".

```
1  > let a = 357;;
2  val a : int = 357
3
4  > let a = 864;;
5  val a : int = 864
```

Listing 5.19 quickStartSumFct.fsx:
A script to add 2 numbers using a user-defined function.

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

```
1  $ fsharpc --nologo quickStartSumFct.fsx && mono
     quickStartSumFct.exe
2  1221
```

code, such that we can focus on the transformation of data by a function while ignoring the details on how this is done. Functions are also useful for code reuse, i.e., instead of repeating a piece of code in several places, such code can be encapsulated in a function and replaced with function calls. This makes debugging and maintenance considerably simpler. Entering the function into an interactive session will illustrate the inferred type the function `sum` has: `val sum : x:int -> y:int -> int`. The "`->`" is the mapping operator in the sense that functions are mappings between sets. The type of the function `sum`, should be read as `val sum : x:int -> (y:int -> int)`, that is, `sum` takes an integer and returns a function, which takes an integer and returns an integer. This is an example of a higher-order function.

Type inference in F# may cause problems, since the type of a function is inferred based on the context in which it is defined. E.g., in an interactive session, defining the sum in one scope on a single line will default the types to integers, F#'s favorite type. Thus, if the next script-fragment uses the function with floats, then we will get an error message as shown in **??**. A remedy is to define the function in the same script-fragment as it is used, such as shown in **??**. Alternatively, the types may be explicitly stated as shown in **??**. The function `sum` has two arguments and a return type, and in **??** we have specified all three. This is done using the "`:`" lexeme, and to resolve confusion, we must use parentheses around the arguments, such as (`y : float`), otherwise F# would not be able to understand whether the

Listing 5.20: Types are inferred in blocks, and F# tends to prefer integers.

```
> let sum x y = x + y;;
val sum : x:int -> y:int -> int

> let c = sum 357.6 863.4;;

   let c = sum 357.6 863.4;;
   ------------^^^^^

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

Listing 5.21: Type inference is per script-fragment.

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

type annotation was for the argument or the return value. Often it is sufficient to specify just some of the types, since type inference will enforce the remaining types. E.g., in this example, the "+" operator is defined for identical types, so specifying the return value of sum to be a float implies that the result of the "+" operator is a float, and therefore that its arguments must be floats, and finally then that the arguments for sum must be floats. However, in this book we advocate the following advice: **specify types unless explicitly working with generic functions.**

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

5.3 Programming as a form of communication

- organisation

Listing 5.22: Function argument and return types may be stated explicitly.

```
1  > let sum (x : float) (y : float) : float = x + y;;
2  val sum : x:float -> y:float -> float
3
4  > let c = sum 357.6 863.4;;
5  val c : float = 1221.0
```

- comments

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 what the code should be doing.

2. Highlight big insights essential for the code.

3. Highlight possible conflicts and/or areas where the code could be changed later.

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.

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

Listing 5.23: 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 5.24: Line comments.

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

where the comment text ends after the first newline.

6 Using F# as a Calculator

In this chapter, we will exclusively use the interactive mode to illustrate basic types and operations in F#.

6.1 Literals and Basic Types

All programs rely on 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 **??**.   What this means

> **Listing 6.1: Typing the number 3.**
>
> ```
> 1  > 3;;
> 2  val it : int = 3
> ```

is that F# has inferred the type to be *int* and bound it to the identifier *it*. For more on binding and identifiers see **??**. 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 **??**.   Each literal represents the number 3, but 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 **??**. 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

Listing 6.2: Many representations of the number 3 but using different types.

```
1   > 3;;
2   val it : int = 3
3
4   > 3.0;;
5   val it : float = 3.0
6
7   > '3';;
8   val it : char = '3'
9
10  > "3";;
11  val it : string = "3"
```

`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 using *scientific notation*, such as `3.5e-4` and `4e2`, where the `e`-notation is translated to a value as $3.5e\text{-}4 = 3.5 \cdot 10^{-4} = 0.00035$, 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 a 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 uses 8 as basis and hexadecimals use 16 as basis. Each octal digit can be represented by exactly three bits, and each hexadecimal digit can 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 a octal number `0o557`, and as a hexadecimal number `0x16f`. In F#, the character sequences `0b12` and `ff` are not recognized as numbers.

Metatype	Type name	Description
Boolean	`bool`	Boolean values true or false
Integer	<u>`int`</u>	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	<u>`float`</u>	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	<u>`char`</u>	Unicode character
	`string`	Unicode sequence of characters
None	<u>`unit`</u>	The value ()
Object	`obj`	An object
Exception	`exn`	An exception

Table 6.1: List of some of the basic types. The most commonly used types are underlined. For a description of integer see Appendix **??**, for floating point numbers see Appendix **??**, for ASCII and Unicode characters see Appendix **??**, for objects see **??**, and for exceptions see **??**.

A *character* is a *Unicode code point*, and character literals are enclosed in single quotation marks. Appendix **??** 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 **??**. Examples of `char` representations of the letter 'a' are: `'a'`, `'\097'`, `'\u0061'`,

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 6.2: Escape characters. The escapecode `\DDD` is sometimes called a tricode.

`'\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 **??**. 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 **??**.

The literal type is closely connected to how the values are represented internally. For example, a value of type `int32` use 32 bits and can be both positive and negative, while a `uint32` value also use 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 precession these types can represent. This is summarized in **??** and discussed in more detail in **??**. 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 6.3: Examples of string literals.

```
1  > "abcde";;
2  val it : string = "abcde"
3
4  > "abc
5  -    de";;
6  val it : string = "abc
7    de"
8
9  > "abc\
10 -    de";;
11 val it : string = "abcde"
12
13 > "abc\nde";;
14 val it : string = "abc
15 de"
```

Many basic types are compatible, and the type of a literal may be changed by *type-casting*. An example of casting to a `float` is shown in **??**. When `float` 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 **??**. Boolean values are often treated as the 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 **??**. 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 **??**.

Typecasting is often a destructive operation, e.g., typecasting a `float` to `int` removes the fractional part without rounding as shown in **??**. Here we typecasted to 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 **??** 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 **??**. I.e., $357.6 + 0.5 = 358.1$ and removing the fractional part by downcasting results in 358, which is the correct answer.

Listing 6.4: Named and implied literals.

```
 1  > 3;;
 2  val it : int = 3
 3
 4  > 4u;;
 5  val it : uint32 = 4u
 6
 7  > 5.6;;
 8  val it : float = 5.6
 9
10  > 7.9f;;
11  val it : float32 = 7.9000001f
12
13  > 'A';;
14  val it : char = 'A'
15
16  > 'B'B;;
17  val it : byte = 66uy
18
19  > "ABC";;
20  val it : string = "ABC"
21
22  > @"abc\nde";;
23  val it : string = "abc\nde"
```

Listing 6.5: Casting an integer to a floating point number.

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

6.2 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 **??**, is

$$\underbrace{357.6}_{\text{operand}} \quad \underbrace{+}_{\text{operator}} \quad \underbrace{0.5}_{\text{operand}} \tag{6.1}$$

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.

Listing 6.6: Casting booleans.

```
1  > System.Convert.ToBoolean 1;;
2  val it : bool = true
3
4  > System.Convert.ToBoolean 0;;
5  val it : bool = false
6
7  > System.Convert.ToInt32 true;;
8  val it : int = 1
9
10 > System.Convert.ToInt32 false;;
11 val it : int = 0
```

Listing 6.7: Fractional part is removed by downcasting.

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

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

Listing 6.9:   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 **??** and **??**, and a range

Listing 6.8: Rounding by modified downcasting.

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

of mathematical functions is shown in **??**.   Note that 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 **??** for more on recursive functions.

Unary operators take only one argument and have the syntax:

Listing 6.10:   A unary expressions.

```
<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 **??**, 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 **??**.   Here, the addition and multipli-

Listing 6.11: A simple arithmetic expression.

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

cation 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 **??**, the precedence is shown by the order they are given in the table.

Associativity describes the order in which calculations are performed for binary operators of the same precedence. Some operator's associativity are given in **??**. 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, association makes a huge difference, e.g., $4^{(3^2)} \neq (4^3)^2$. Examples of this is shown in **??**. **Whenever in doubt of association or any other basic**

Listing 6.12: Precedence rules define implicit parentheses.

```
1   > 4.0 * 3.0 * 2.0;;
2   val it : float = 24.0
3
4   > (4.0 * 3.0) * 2.0;;
5   val it : float = 24.0
6
7   > 4.0 * (3.0 * 2.0);;
8   val it : float = 24.0
9
10  > 4.0 ** 3.0 ** 2.0;;
11  val it : float = 262144.0
12
13  > (4.0 ** 3.0) ** 2.0;;
14  val it : float = 4096.0
15
16  > 4.0 ** (3.0 ** 2.0);;
17  val it : float = 262144.0
```

---

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.

6.3 Boolean Arithmetic

Boolean arithmetic is the basis of almost all computers and particularly important for controlling program flow, which will be discussed in **??**. 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 combination 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 **??**. A good mnemonic for remembering the result of the *'and'* and *'or'* operators is to 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 **??**. Here, we used the `printfn` function to

Listing 6.13: Boolean operators and truth tables.

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

present the results of many expressions on something that resembles a tabular form. The spacing produced using the `printfn` function is not elegant, and in Section 2.3 we will discuss better options for producing more beautiful output. Notice that the arguments for `printfn` was given on the next line with indentation. The indentation is an important part of telling F# which part of what you write belong 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 **??**.

6.4 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 **??** are finite subsets reduced by limiting their ranges. An in-depth description of integer implementation can be found in **??**. The type `int` is the most common type.

?????? 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 specified using the

"y"-literal and can hold values $[-128 \ldots 127]$. This causes problems in the example in
??. Here $100 + 30 = 130$, which is larger than the biggest `sbyte`, and the result is an

---

Listing 6.14: Adding integers may cause overflow.

```
1  > 100y;;
2  val it : sbyte = 100y
3
4  > 30y;;
5  val it : sbyte = 30y
6
7  > 100y + 30y;;
8  val it : sbyte = -126y
```

---

overflow. Similarly, we get an underflow, when the arithmetic result falls below the
smallest value storable in an `sbyte`, as demonstrated in **??**. I.e., we were expecting

---

Listing 6.15: Subtracting integers may cause underflow.

```
1  > -100y - 30y;;
2  val it : sbyte = 126y
```

---

a negative number but got a positive number instead.

The overflow error in **??** 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 **??**. That is, for signed bytes, the left-most bit is used to

---

**Listing 6.16: The leftmost bit is interpreted differently for signed and
unsigned integers, which gives rise to potential overflow errors.**

```
1  > 0b10000010uy;;
2  val it : byte = 130uy
3
4  > 0b10000010y;;
5  val it : sbyte = -126y
```

---

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 **??**. To-

---

Listing 6.17: Integer division and remainder operators.

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

---

gether, the integer division and remainder can form a lossless representation of the original number, see **??**. Here we see that integer division of 7 by 3 followed by

---

Listing 6.18: Integer division and remainder is a lossless representation of an integer, compare with ??.

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

---

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

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

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

Listing 6.19: 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@ () [0x00000]
     in <ea55944aeec94f51b27071fca4652afa >:0
4    at (wrapper managed -to-native)
     System.Reflection.RuntimeMethodInfo.InternalInvoke(System.Reflection
5    at System.Reflection.RuntimeMethodInfo.Invoke
     (System.Object obj, System.Reflection.BindingFlags
     invokeAttr, System.Reflection.Binder binder,
     System.Object[] parameters,
     System.Globalization.CultureInfo culture) [0x0006a] in
     <e068e2227ab74c1bb3d724ebaab0e3ff >:0
6  Stopped due to error
```

Listing 6.20: Integer exponent function.

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

'or' operator position-wise; and `<expr> ^^^ <expr>` returns the result of the Boolean 'xor' operator defined by the truth table in **??**.

6.5 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 **??**. The type `float` is the most common type.

?????? 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 **??**. The remainder for 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 **??**.

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 **??**, no exception is thrown. However, the `float` type has limited precision, since there is

Listing 6.21: Floating point division and remainder operators.

```
1  > 7.0 / 2.5;;
2  val it : float = 2.8
3
4  > 7.0 % 2.5;;
5  val it : float = 2.0
```

Listing 6.22: Floating point division, downcasting, and remainder is a lossless representation of a number.

```
1  > float (int (7.0 / 2.5));;
2  val it : float = 2.0
3
4  > (float (int (7.0 / 2.5))) * 2.5;;
5  val it : float = 5.0
6
7  > (float (int (7.0 / 2.5))) * 2.5 + 7.0 % 2.5;;
8  val it : float = 7.0
```

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 **??**. 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 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**

Listing 6.23: Floating point numbers include infinity and Not-a-Number.

```
1  > 1.0/0.0;;
2  val it : float = infinity
3
4  > 0.0/0.0;;
5  val it : float = nan
```

Listing 6.24: Floating point arithmetic has finite precision.

```
1  > 357.8 + 0.1 - 357.9;;
2  val it : float = 5.684341886e-14
```

---

as, abs ((357.8 + 0.1) - 357.9) < 1e-10.

6.6 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 **??**. I.e., the code

---

Listing 6.25: Converting case by casting and integer arithmetic.

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

---

point difference between upper and lower case for any alphabetical character 'a' to 'z' is constant, hence we can change 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 **??**. Characters and strings

---

Listing 6.26: Example of string concatenation.

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

---

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 be indexed as using the *.[]* notation. This is demonstrated in **??**. Notice that the first character has index 0, and to get the last character in a string, we use the string's `length` property. This is done as shown in **??**. Since 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

---

Listing 6.27: String indexing using square brackets.

```
1  > "abcdefg".[0];;
2  val it : char = 'a'
3
4  > "abcdefg".[3];;
5  val it : char = 'd'
6
7  > "abcdefg".[3..];;
8  val it : string = "defg"
9
10 > "abcdefg".[..3];;
11 val it : string = "abcd"
12
13 > "abcdefg".[1..3];;
14 val it : string = "bcd"
15
16 > "abcdefg".[*];;
17 val it : string = "abcdefg"
```

Listing 6.28: String length attribute and string indexing.

```
1  > "abcdefg".Length;;
2  val it : int = 7
3
4  > "abcdefg".[7-1];;
5  val it : char = 'g'
```

---

built-in functions in `System.String` for working with strings, some of which will be discussed in **??**.

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 **??**.

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.

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 6.3: 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 **??** for details.

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 6.4: 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.

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 6.5: 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 6.6: Predefined functions for arithmetic operations.

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 6.7: 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..

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

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

Table 6.9: Boolean exclusive or truth table.

6.7 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 6.29:   Tuples are list of expressions separated by commas.

1    <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 **??**. The values 2, `true`,

```
Listing 6.30: Tuple types are products of sets.

1    > let tp = (2, true, "hello")
2    - printfn "%A" tp;;
3    (2, true, "hello")
4    val tp : int * bool * string = (2, true, "hello")
5    val it : unit = ()
```

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 types 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 **??**. 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 and will be discussed in further details in **??**. Since we used the `\%A` placeholder in the `printfn` function, the function can be 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 **??** 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 **??**.

Tuples of equal lengths can be compared, and the comparison is defined similarly

---

Listing 6.31: Definition of a tuple.

```
1  > let deconstructNPrint tp =
2  -     let (a, b, c) = tp
3  -     printfn "tp = (%A, %A, %A)" a b c
4  -
5  - deconstructNPrint (2, true, "hello")
6  - deconstructNPrint (3.14, "Pi", 'p');;
7  tp = (2, true, "hello")
8  tp = (3.14, "Pi", 'p')
9  val deconstructNPrint : 'a * 'b * 'c -> unit
10 val it : unit = ()
```

Listing 6.32 pair.fsx:
Deconstruction of pairs with the built-in functions `fst` and `snd`.

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

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

---

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 **??** 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 **??**. 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 **??**. Mutable tuples are value types, meaning that binding to new names makes copies, not aliases, as demonstrated in **??**. The use of tuples shortens code and highlights semantic 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

Listing 6.33 tupleCompare.fsx:
Tuples comparison is similar to string comparison.

```fsharp
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
```

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.**

6.8 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

---

Listing 6.34 tupleOfMutables.fsx:
A mutable changes value, but the tuple defined by it does not refer to
the new value.

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

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

Listing 6.35 mutableTuple.fsx:
A mutable tuple can be assigned a new value.

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

```
1  $ fsharpc --nologo mutableTuple.fsx && mono
      mutableTuple.exe
2  (1, 2)
3  (3, 4)
```

---

decimal value is calculated as,

$$v = \sum_{i=0}^{n} b_i 2^i \tag{6.2}$$

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 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

---

Listing 6.36 mutableTupleValue.fsx:
A mutable tuple is a value type.

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

```
1  $ fsharpc --nologo mutableTupleValue.fsx && mono
     mutableTupleValue.exe
2  (3, 4) (1, 2)
```

---

the number 11_{10} in decimal form to binary form, we would perform the following steps:

$$11 \text{ div } 2 = 5, \ 11 \text{ rem } 2 = 1$$

$$5 \text{ div } 2 = 2, \ 5 \text{ rem } 2 = 1$$

$$2 \text{ div } 2 = 1, \ 2 \text{ rem } 2 = 0$$

$$1 \text{ div } 2 = 0, \ 1 \text{ rem } 2 = 1$$

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 6.37: Converting the number 11_{10} to binary form.**
>
> ```
> 1  > printfn "(%d, %d)" (11 / 2) (11 % 2);;
> 2  (5, 1)
> 3  val it : unit = ()
> 4  > printfn "(%d, %d)" (5 / 2) (5 % 2);;
> 5  (2, 1)
> 6  val it : unit = ()
> 7  > printfn "(%d, %d)" (2 / 2) (2 % 2);;
> 8  (1, 0)
> 9  val it : unit = ()
> 10 > printfn "(%d, %d)" (1 / 2) (1 % 2);;
> 11 (0, 1)
> 12 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.

7 Values, Functions, and Statements

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,

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

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \qquad (7.4)$$

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 $\pm$ in the equation. Details on function definition is given in **??**. Our solution thus looks like **??**. 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

> ### Listing 7.1 identifiersExample.fsx:
> ### Finding roots for quadratic equations using function name binding.
>
> ```
> 1  let discriminant a b c = b ** 2.0 - 4.0 * a * c
> 2  let positiveSolution a b c = (-b + sqrt (discriminant a b c)) /
>       (2.0 * a)
> 3  let negativeSolution a b c = (-b - sqrt (discriminant a b c)) /
>       (2.0 * a)
> 4
> 5  let a = 1.0
> 6  let b = 0.0
> 7  let c = -1.0
> 8  let d = discriminant a b c
> 9  let xp = positiveSolution a b c
> 10 let xn = negativeSolution a b c
> 11 do printfn "0 = %A * x ** 2.0 + %A * x + %A" a b c
> 12 do printfn "  has discriminant %A and solutions %A and %A" d xn xp
> ```
>
> ```
> 1  $ fsharpc --nologo identifiersExample.fsx && mono
>       identifiersExample.exe
> 2  0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
> 3    has discriminant 4.0 and solutions -1.0 and 1.0
> ```

avoiding possible typing mistakes and reducing the amount of code which 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

- 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 of letters, digits, and a range of special characters except for SP, LF, and CR (space, line feed, and carriage return). See **??** for more on codepoints that represents letters.

- They can also be a sequence of identifiers separated by a period.

- They cannot be keywords, see **??**.

Examples of identifiers are: `a`, `theCharacter9`, `Next_Word`, `_tok`, and `f.sharp.rocks`. Since programmers often work in multilingual environment dominated by the English language it is advicable to **restrict identifiers to use letters from the English**

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

alphabet, numbers, period, and '_'. However, the number of possible identifiers is enormous. The full definition refers to the Unicode general categories described in Appendix **??**, and there are currently 19.345 possible Unicode code points in the letter 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 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 been chosen to be discriminant. F# places no special significance to 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 confuse 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 **??**. 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 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 character, 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 $f\,3$, 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 **????**.

- They can be `do`-bindings that produce side-effects and whose result are ignored, see **??**

- They can be assignments to variables, see **??**.

- 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,
but the syntaxes may be mixed, and we will highlight the options, when relevant.

7.1 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 7.2:   Value binding expression.**
>
> ```
> 1  let <valueIdent> = <bodyExpr> [in <expr>]
> ```

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 following square bracket notation `[]` means
that the enclosed is optional, and F# is able to identify whether or not the optional
part is used as signified by the optional presence of the `in` keyword. If the `in` keyword
is used, then the value-identifier is a local definition in the `<expr>` expression, and it
is not available in later lines. For lightweight syntax, the `in` keyword is replaced with
a newline, and the binding *is* available in later lines until the end of the scope it is
defined in.

The value identifier 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. See **??** for more details on patterns.

For example, letting the identifier `p` be bound to the value `2.0` and using it in an
expression is done as shown in **??**. F# will ignore most newlines between lexemes,
i.e., the above is equivalent to writing as shown in **??**. F# also allows for an
alternative notation called *lightweight syntax*, where e.g., the `in` keyword is replaced
with a newline, and the expression starts on the next line at the same column as `let`
starts in, i.e., the above is equivalent to **??**. The same expression in interactive mode
will also show with the inferred types, as shown in **??**. By 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

> **Listing 7.3 letValue.fsx:**
> The identifier `p` is used in the expression following the `in` keyword.
>
> ```fsharp
> 1  let p = 2.0 in do printfn "%A" (3.0 ** p)
> ```
> ---
> ```
> 1  $ fsharpc --nologo letValue.fsx && mono letValue.exe
> 2  9.0
> ```

> **Listing 7.4 letValueLF.fsx:**
> Newlines after `in` make the program easier to read.
>
> ```fsharp
> 1  let p = 2.0 in
> 2  do printfn "%A" (3.0 ** p)
> ```
> ---
> ```
> 1  $ fsharpc --nologo letValueLF.fsx && mono letValueLF.exe
> 2  9.0
> ```

types gives an error, as shown in **??**. 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.

An expression can be a sequence of expressions separated by the lexeme "`;`", see **??**. The lightweight syntax automatically inserts the "`;`" lexeme at newlines, hence using the lightweight syntax, the above is the same as shown in **??**.

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 higher scopes, see **??** for an example. This is called *lexical scope*. Some special

> **Listing 7.5 letValueLightWeight.fsx:**
> Lightweight syntax does not require the `in` keyword, but the expression must be aligned with the `let` keyword.
>
> ```fsharp
> 1  let p = 2.0
> 2  do printfn "%A" (3.0 ** p)
> ```
> ---
> ```
> 1  $ fsharpc --nologo letValueLightWeight.fsx && mono
>      letValueLightWeight.exe
> 2  9.0
> ```

Listing 7.6: Interactive mode also outputs inferred types.

```
1  > let p = 2.0
2  - do printfn "%A" (3.0 ** p);;
3  9.0
4  val p : float = 2.0
5  val it : unit = ()
```

Listing 7.7 letValueTypeError.fsx:
Binding error due to type mismatch.

```
1  let p : float = 3
2  do printfn "%A" (3.0 ** p)
```
- -
```
1  $ fsharpc --nologo letValueTypeError.fsx && mono
     letValueTypeError.exe
2
3  letValueTypeError.fsx(1,17): error FS0001: This expression
     was expected to have type
4      'float'
5  but here has type
6      'int'
```

---

bindings are mutable, in which case F# uses the *dynamic scope*, that is, the value of a binding is defined by when it is used. This will be discussed in **??**.

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 **??**. However, using parentheses, we create a *code block*, i.e., a *nested scope*, and then redefining is allowed, as demonstrated in **??**. Nevertheless, **avoid reusing names unless it's in a deeper scope.**

Inside the block in **??** we used indentation, which is good practice, but not required here.

Bindings inside a nested scope are not available outside, as shown in **??**. Nesting is a natural part of structuring code, e.g., through function definitions to be discussed in **??** and flow control structures to be discussed in **??**. 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.

Listing 7.8 letValueSequence.fsx:
A value-binding for a sequence of expressions.

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

```
1  $ fsharpc --nologo letValueSequence.fsx && mono
     letValueSequence.exe
2  2.0
3  9.0
```

Listing 7.9 letValueSequenceLightWeight.fsx:
A value-binding for a sequence using lightweight syntax.

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

```
1  $ fsharpc --nologo letValueSequenceLightWeight.fsx && mono
     letValueSequenceLightWeight.exe
2  2.0
3  9.0
```

Defining blocks is used for controlling the extent of a lexical scope of bindings. For example, adding a second `printfn` statement, as in **??**, will print the value 4, last bound to the identifier p, since F# interprets the above as `let p = 3 in let p = 4 in (printfn "%A" p; printfn "%A" p)`. Had we intended to print the two different values of p, then we should have created a block as in **??**.

Listing 7.10 letValueScopeLower.fsx:
Redefining identifiers is allowed in lower scopes.

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

```
1  $ fsharpc --nologo letValueScopeLower.fsx && mono
     letValueScopeLower.exe
2    4
```

Listing 7.11 letValueScopeLowerError.fsx:
Redefining identifiers is not allowed in lightweight syntax at top level.

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

```
1  $ fsharpc --nologo -a letValueScopeLowerError.fsx
2
3  letValueScopeLowerError.fsx(2,5): error FS0037: Duplicate
     definition of value 'p'
```

Listing 7.12 letValueScopeBlockAlternative3.fsx:
A block may be created using parentheses.

```
1  (
2    let p = 3
3    let p = 4
4    do printfn "%A" p
5  )
```

```
1  $ fsharpc --nologo letValueScopeBlockAlternative3.fsx &&
     mono letValueScopeBlockAlternative3.exe
2  4
```

Listing 7.13 letValueScopeNestedScope.fsx:
Bindings inside a scope are not available outside.

```
1  let p = 3
2  (
3    let q = 4
4    do printfn "%A" q
5  )
6  do printfn "%A %A" p q
```

```
1  $ fsharpc --nologo -a letValueScopeNestedScope.fsx
2
3  letValueScopeNestedScope.fsx(6,22): error FS0039: The
     value or constructor 'q' is not defined.
```

Listing 7.14 letValueScopeBlockProblem.fsx:
Overshadowing hides the first binding.

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

```
1  $ fsharpc --nologo letValueScopeBlockProblem.fsx && mono
     letValueScopeBlockProblem.exe
2  4
3  4
```

Listing 7.15 letValueScopeBlock.fsx:
Blocks allow for the return to the previous scope.

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

```
1  $ fsharpc --nologo letValueScopeBlock.fsx && mono
     letValueScopeBlock.exe
2  4
3  3
```

7.2 Function Bindings

A function is a mapping between an input and output domain. A key advantage of using functions when programming is that they encapsulate code into smaller units, that are easier to debug and may be reused. 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 7.16:   Function binding expression

let <funcIdent> <arg> {<arg>} | () = <bodyExpr> [in <expr>]
```

Here `<funcIdent>` 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 `<bodyExpr>`. 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 7.17:   Function binding expression

let <funcIdent> (<arg> : <type>) {(<arg> : <type>)} : <type>
    | () : <type> = <bodyExpr> [in <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 **??** and higher-order functions in **??**.

An example of defining a function and using it in interactive mode is shown in **??**. Here we see that the function is 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 argument 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

---

Listing 7.18: An example of a binding of an identifier and a function.

```
1  > let sum (x : float) (y : float) : float = x + y in
2  - let c = sum 357.6 863.4 in
3  - do printfn "%A" c;;
4  1221.0
5  val sum : x:float -> y:float -> float
6  val c : float = 1221.0
7  val it : unit = ()
```

---

have declare the type of the result, as in **??**.   Or even just one of the arguments,

---

Listing 7.19 letFunctionAlterantive.fsx:
Not every type needs to be declared.

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

---

as in **??**.   In both cases, since the + *operator* is only defined for *operands* of the

---

Listing 7.20 letFunctionAlterantive2.fsx:
Just one type is often enough for F# to infer the rest.

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

---

same type, declaring the type of either arguments or result implies the type of the remainder. As for values, lightweight syntax automatically inserts the keyword `in` and the lexeme ";", as shown in **??**.

Arguments need not always be inferred to types, but may be of the generic type when *type safety* is ensured, as shown in **??**.   Here, the function `second` 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 **??**.   Here, we used the lightweight syntax, where the "=" identifies the start of a nested scope, and F# identifies the scope by 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

---

Listing 7.21 letFunctionLightWeight.fsx:
Lightweight syntax for function definitions.

```
1  let sum x y : float = x + y
2  let c = sum 357.6 863.4
3  do printfn "%A" c
```

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

Listing 7.22: Type safety implies that a function will work for any
type.

```
1  > let second x y = y
2  - let a = second 3 5
3  - do printfn "%A" a
4  - let b = second "horse" 5.0
5  - do printfn "%A" b;;
6  5
7  5.0
8  val second : x:'a -> y:'b -> 'b
9  val a : int = 5
10 val b : float = 5.0
11 val it : unit = ()
```

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 becausethe scope ends before `let a = 1.0`, `discriminant` cannot be called outside `solution`.

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

> **Listing 7.23 identifiersExampleAdvance.fsx:**
> **A function may contain sequences of expressions.**
>
> ```
> 1  let solution a b c sgn =
> 2    let discriminant a b c =
> 3      b ** 2.0 - 4.0 * a * c
> 4    let d = discriminant a b c
> 5    (-b + sgn * sqrt d) / (2.0 * a)
> 6
> 7  let a = 1.0
> 8  let b = 0.0
> 9  let c = -1.0
> 10 let xp = solution a b c +1.0
> 11 let xn = solution a b c -1.0
> 12 do printfn "0 = %A * x ** 2.0 + %A * x + %A" a b c
> 13 do printfn "  has solutions %A and %A" xn xp
> ```
> ```
> 1  $ fsharpc --nologo identifiersExampleAdvance.fsx && mono
>      identifiersExampleAdvance.exe
> 2  0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
> 3    has solutions -1.0 and 1.0
> ```

Functions do not need a name, but may be declared as an *anonymous function* using the *fun* keyword and the *"->"* lexeme, as shown in **??**. Here, a name is bound to an anonymous function which returns the first of two arguments. 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 **??**. A common use of anonymous functions is as arguments to other functions, as demonstrated in **??**. Note that here `apply` is given 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 **??**. In the example we combine 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 **??**. The example shows regular composition, left-to-right, and right-to-left piping. The word piping is a picturial 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 **??** perform the same calculation. The

---

Listing 7.24 lexicalScopeNFunction.fsx:
Lexical scope means that $f(z) = 3x$ and not $4x$ at the time of calling.

```
1  let testScope x =
2    let a = 3.0
3    let f z = a * z
4    let a = 4.0
5    f x
6  do printfn "%A" (testScope 2.0)
```
- -
```
1  $ fsharpc --nologo lexicalScopeNFunction.fsx && mono
     lexicalScopeNFunction.exe
2  6.0
```

Listing 7.25 functionDeclarationAnonymous.fsx:
Anonymous functions are functions as values.

```
1  let first = fun x y -> x
2  do printfn "%d" (first 5 3)
```
- -
```
1  $ fsharpc --nologo functionDeclarationAnonymous.fsx &&
     mono functionDeclarationAnonymous.exe
2  5
```

---

left-to-right piping in line **??** 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 **??** has the order of arithmetic composition as line **??**. Unfortunately, since the piping operators are left-associatitve, without the parenthesis in line **??** `g <| f <| 2`, F# would read the expression as `(g <| f) <| 2`. That would have been an error, since `g` takes an integer as argument, not a function. F# can also define composition on a function level. Further discussion on this is deferred to **??**. The piping operator comes in four variants: "`||>`", "`<||`", "`|||>`", and "`<|||`". These allow for piping between pairs and triples to functions of 2 and 3 arguments, see **??** for an example.   The example demonstrates right-to-left piping, left-to-right works analogously.

A *procedure* is a generalization of the concept of functions, and in contrast to functions, procedures need not return values. This is demonstrated in **??**.   In F#, this is automatically given the unit type as the return value. Procedural thinking is useful for *encapsulation* of scripts, but is prone to *side-effects*. For this reason, it is adviced to **prefer functions over procedures.** More on side-effects in **??**.

Listing 7.26 functionDeclarationAnonymousAdvanced.fsx:
Anonymous functions are often used as arguments for other functions.

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

```
1  $ fsharpc --nologo
       functionDeclarationAnonymousAdvanced.fsx && mono
       functionDeclarationAnonymousAdvanced.exe
2  18
```

Listing 7.27 functionComposition.fsx:
Composing functions using intermediate bindings.

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

```
1  $ fsharpc --nologo functionComposition.fsx && mono
       functionComposition.exe
2  a = 3, b = 9, c = 9
```

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

$$\texttt{mul} : (\text{args}, \exp, \text{env}) = \big((\texttt{x,y}), (\texttt{x * y}), ()\big) \tag{7.5}$$

$$\texttt{applyFactor} : (\text{args}, \exp, \text{env}) = \big((\texttt{x, fct}), (\texttt{body}), (\texttt{factor} \to 2.0)\big) \tag{7.6}$$

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 are values

Listing 7.28 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  $ fsharpc --nologo functionPiping.fsx && mono
     functionPiping.exe
2  a = 9, b = 9, c = 9
```

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 **??** line **??**, 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 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.

Listing 7.29 functionTuplePiping.fsx:
Tuples can be piped to functions of more than one argument.

```
1  let f x = printfn "%A" x
2  let g x y = printfn "%A %A" x y
3  let h x y z = printfn "%A %A %A" x y z
4
5  1 |> f
6  (1, 2) ||> g
7  (1, 2, 3) |||> h
```

```
1  $ fsharpc --nologo functionTuplePiping.fsx && mono
     functionTuplePiping.exe
2  1
3  1 2
4  1 2 3
```

Listing 7.30 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  $ fsharpc --nologo procedure.fsx && mono procedure.exe
2  This is '3'
3  This is '3.0'
```

Listing 7.31 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  $ fsharpc --nologo functionFirstClass.fsx && mono
     functionFirstClass.exe
2  15
3  6
```

7.3 Operators

Operators are functions, and in F#, the infix multiplication operator + is equivalent to the function (+), as shown in **??**. All operators have this option, and you

---

Listing 7.32 addOperatorNFunction.fsx:
Operators have function equivalents.

```
1  let a = 3.0
2  let b = 4.0
3  let c = a + b
4  let d = (+) a b
5  do printfn "%A plus %A is %A and %A" a b c d
```

```
1  $ fsharpc --nologo addOperatorNFunction.fsx && mono
     addOperatorNFunction.exe
2  3.0 plus 4.0 is 7.0 and 7.0
```

---

may redefine them and define your own operators, but in F# names of user-defined operators are limited:

- A *unary operator* name can be: "+", "-", "+.", "-.", "%", "&", "&&", "~~", "~~~", "~~~~", ..., apostropheOp. Here apostropheOp is an operator name starting with "!" and followed by one or more of either "!", "%", "&", "*", "+", "-", ".", "/", "<", "=", ">", "@", "^", "|", "~", but apostropheOp cannot be "!=".

- An *binary operator* name can be: "+", "-", "+.", "-.", "%", "&", "&&", ":=", "::", "$", "?", dotOp. Here dotOp is an operator name starting with "." and followed by "+", "-", "+.", "-.", "%", "&", "&&", "-", "+", "||", "<", ">", "=", "|", "&", "^", "*", "/", "%", "!=". Only "?" and "?<-" may start with "?".

The precedence rules and associativity of user-defined operators follow the rules for which they share prefixes with built-in rules, see **??**. For example, .*, +++, and <+ are valid operator names for infix operators, they have precedence as ordered, and their associativities are all left. Using ~ as the first character in the definition of an operator makes the operator unary and will not be part of the name. Examples of definitions and use of operators are, Operators beginning with * must use a space in its definition. For example, without a space (* would be confused with the beginning of a comment (*, see **??** for more on comments in the code.

Beware, redefining existing operators lexically redefines all future uses of the operators for all types, hence **it is not a good idea to redefine operators, but better to define new ones.** In **??** we will discuss how to define type-specific operators,

Listing 7.33 operatorDefinitions.fsx:
Operators may be (re)defined by their function equivalent.

```
1  let (.*) x y = x * y + 1
2  printfn "%A" (3 .* 4)
3  let (+++) x y = x * y + y
4  printfn "%A" (3 +++ 4)
5  let (<+) x y = x < y + 2.0
6  printfn "%A" (3.0 <+ 4.0)
7  let (~+.) x = x+1
8  printfn "%A" (+.1)
```

```
1  $ fsharpc --nologo operatorDefinitions.fsx && mono
     operatorDefinitions.exe
2  13
3  16
4  true
5  2
```

including prefix operators.

7.4 Do-Bindings

Aside from `let`-bindings that binds 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 7.34: Syntax for do-bindings.

```
1  [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 are the `printf` family described below. In the remainder of this book, we will refrain from using the `do` keyword.

8 In-code Documentation

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 what the code should be doing.

2. Highlight big insights essential for the code.

3. Highlight possible conflicts and/or areas where the code could be changed later.

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.

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

Listing 8.1: 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 8.2: Line comments.

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

where the comment text ends after the first newline.

The F# compiler has an option for generating *Extensible Markup Language* (*XML*) files from scripts using the C# documentation comments tags[1]. The XML documentation starts with a triple-slash ///, i.e., a lineComment and a slash, which serve as comments for the code construct that follows immediately after. XML consists of tags which always appear in pairs, e.g., the tag "tag" would look like `<tag> ... </tag>`. F# accept any tags, but recommends those listed in **??**. If no tags are used, then it

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

is automatically assumed to be a `<summary>`. An example of a documented script is shown in **??**. is:

Mono's `fsharpc` command may be used to extract the comments into an XML file, as demonstrated in **??**.

Listing 8.4, Converting in-code comments to XML.

```
1  $ fsharpc --doc:commentExample.xml commentExample.fsx
2  F# Compiler for F# 4.0 (Open Source Edition)
3  Freely distributed under the Apache 2.0 Open Source License
```

---

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

Listing 8.3 commentExample.fsx:
Code with XML comments.

```fsharp
/// 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 a = 1.0
///      let b = 0.0
///      let c = -1.0
///      let xp = (solution a b c +1.0)
///      printfn "0=%.1fx^2+%.1fx+%.1f => x_+=%.1f" a b c xp
///   </code>
///   prints <c>0=1.0x^2+0.0x+-1.0 => x_+=0.7</c>.
/// </example>
/// <param name="a">Quadratic coefficient.</param>
/// <param name="b">Linear coefficient.</param>
/// <param name="c">Constant coefficient.</param>
/// <param name="sgn">+1 or -1 determines the solution.</param>
/// <returns>The solution to x.</returns>
let solution a b c sgn =
  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)
printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
```

This results in an XML file with the content shown in **??**.

Listing 8.5, An XML file generated by `fsharpc`.

```
1   <?xml version="1.0" encoding="utf-8"?>
2   <doc>
3   <assembly><name>commentExample</name></assembly>
4   <members>
5   <member name="M:CommentExample.solution(System.Double,System.
       Double,System.Double,System.Double)">
6    <summary>Find x when 0 = ax^2+bx+c.</summary>
7    <remarks>Negative discriminants are not checked.</remarks>
8    <example>
9      The following code:
10     <code>
11       let a = 1.0
12       let b = 0.0
13       let c = -1.0
14       let xp = (solution a b c +1.0)
15       printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
16     </code>
17     prints <c>0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7</c> to the
       console.
18   </example>
19   <param name="a">Quadratic coefficient.</param>
20   <param name="b">Linear coefficient.</param>
21   <param name="c">Constant coefficient.</param>
22   <param name="sgn">+1 or -1 determines the solution.</param>
23   <returns>The solution to x.</returns>
24   </member>
25   <member name="M:CommentExample.discriminant(System.Double,System.
       Double,System.Double)">
26   <summary>
27    The discriminant of a quadratic equation with parameters a, b,
       and c
28   </summary>
29   </member>
30   </members>
31   </doc>
```

The extracted XML is written in C# type by convention, since F# is part of the Mono and .Net framework that may be used by any of the languages using Assemblies. Besides the XML inserted in the script, the XML has added the `<?xml ...>` header, `<doc>`, `<assembly>`, `<members>`, and `<member>` tags. The header and the `<doc>` tag are standards for XML. The extracted XML is geared towards documenting big libraries of codes and thus highlights the structured programming organisation, see ????, and `<assembly>`, `<members>`, and `<member>` are indications for where the functions belong in the hierarchy. As an example, the prefix `M:CommentExample.` indicates that the method is in the namespace `commentExample`, which in this case is the name of the file. Furthermore, the function type

```
val solution : a:float->b:float->c:float->sgn:float->float
```

is in the XML documentation

M:CommentExample.solution(System.Double,System.Double,System.Double,System.Double),

which is the C# equivalent.

An accompanying program in the Mono suite is `mdoc`, whose primary use is to perform a syntax analysis of an assembly and generate a scaffold XML structure for an accompanying document. With the `-i` flag, it is further possible to include the in-code comments as initial descriptions in the XML. The XML may be updated gracefully by `mdoc` as the code develops, without destroying manually entered documentation in the accompanying documentation. Finally, the XML may be exported to HTML.

The primary use of the `mdoc` command is to analyze compiled code and generate an empty XML structure with placeholders to describe functions, values, and variables. This structure can be updated and edited as the program develops, and the edited XML files can be exported to *Hyper Text Markup Language* (*HTML*) files and viewed in any browser. Using the console, all of this is accomplished by the procedure shown in **??**, and the result is shown in **??**.

Listing 8.6, Converting an XML file to HTML.

```
1  $ mdoc update -o commentExample -i commentExample.xml
     commentExample.exe
2  New Type: CommentExample
3  Member Added: public static double determinant (double a,
     double b, double c);
4  Member Added: public static double solution (double a,
     double b, double c, double sgn);
5  Member Added: public static double a { get; }
6  Member Added: public static double b { get; }
7  Member Added: public static double c { get; }
8  Member Added: public static double xp { get; }
9  Namespace Directory Created:
10 New Namespace File:
11 Members Added: 6, Members Deleted: 0
12 $ mdoc export-html -out commentExampleHTML commentExample
13 .CommentExample
```

A full description of how to use `mdoc` is found here[2].

---

[2]`http://www.mono-project.com/docs/tools+libraries/tools/monodoc/`
`generating-documentation/`

solution Method

Find x when 0 = ax^2+bx+c.

Syntax

> [Microsoft.FSharp.Core.CompilationArgumentCounts(Mono.Cecil.CustomAttributeArgument[])]
> public static double **solution** (double a, double b, double c, double sgn)

Parameters

a
    Quadratic coefficient.
b
    Linear coefficient.
c
    Constant coefficient.
sgn
    +1 or -1 determines the solution.

Returns

The solution to x.

Remarks

Negative discriminants are not checked.

Example

The following code:

```
Example
    let a = 1.0
    let b = 0.0
    let c = -1.0
    let xp = (solution a b c +1.0)
    printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
```

prints `0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7` to the console.

Requirements

Namespace:
Assembly: commentExample (in commentExample.dll)
Assembly Versions: 0.0.0.0

Figure 8.1: Part of the HTML documentation as produced by `mdoc` and viewed in a browser.

9 Controlling Program Flow

Non-recursive functions encapsulate code and allow for control of execution flow. That is, if a piece of code needs to be executed many times, then we can encapsulate it in the body of a function and call this function several times. In this chapter, we will look at more general control of flow via loops and conditional execution. Recursion is another mechanism for controlling flow, but this is deferred to **??**.

9.1 While and For Loops

Many programming constructs need to be repeated, and F# contains many structures for repetition. A *while*-loop has the following syntax:

```
Listing 9.1:  While loop.
while <condition> do <expr> [done]
```

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. Using lightweight syntax, the block following the *do* keyword up to and including the *done* keyword may be replaced by a newline and indentation.

The program in **??** is an example of a while-loop which counts from 1 to 10. The

```
Listing 9.2 countWhile.fsx:
Count to 10 with a counter variable.

let mutable i = 1 in while i <= 10 do printf "%d " i; i <- i + 1
    done;
printf "\n"
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
$ fsharpc --nologo countWhile.fsx && mono countWhile.exe
1 2 3 4 5 6 7 8 9 10
```

variable `i` is customarily called the counter variable. The counting is done by performing the following computation: In line **??**, the counter variable is first given an initial value of 1. Then execution enters the while-loop and examines the condition. Since $1 <= 10$, the condition is true, and execution enters the body of the loop. The body prints the value of the counter to the screen and increases the counter by 1. Then execution returns to the top of the while-loop. 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 **??**.

In lightweight syntax, this would be as shown in **??**. Notice that although the

Listing 9.3 countWhileLightweight.fsx:
Count to 10 with a counter variable using lightweight syntax.

```
1  let mutable i = 1
2  while i <= 10 do
3    printf "%d " i
4    i <- i + 1
5  printf "\n"
```

```
1  $ fsharpc --nologo countWhileLightweight.fsx && mono
     countWhileLightweight.exe
2  1 2 3 4 5 6 7 8 9 10
```

expression following the condition is preceded with a `do` keyword, and `do <expr>` is a `do`-binding, the keyword `do` is mandatory.

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 9.4:   For loop.

```
1  for <ident> = <firstExpr> to <lastExpr> do <bodyExpr> [done]
```

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>`. As for while-loops, when using lightweight syntax the block following the *do* keyword up to and including the *done* keyword may be replaced by a newline and indentation.

The counting example from **??** using a `for`-loop is shown in **??** As this interactive

---

Listing 9.5 count.fsx:
Counting from 1 to 10 using a `for`-loop.

```
1  for i = 1 to 10 do printf "%d " i done
2  printfn ""
```
```
1  $ fsharpc --nologo count.fsx && mono count.exe
2  1 2 3 4 5 6 7 8 9 10
```

---

script demonstrates, the identifier `i` takes all the values between 1 and 10, but in spite of its changing state, it is not mutable. Note also that the return value of the `for` expression is "()", like the `printf` functions. The lightweight equivalent is shown in **??**.

Listing 9.6 countLightweight.fsx:
Counting from 1 to 10 using a `for`-loop using the lightweight syntax.

```
1  for i = 1 to 10 do
2      printf "%d " i
3  printfn ""
```
```
1  $ fsharpc --nologo countLightweight.fsx && mono
      countLightweight.exe
2  1 2 3 4 5 6 7 8 9 10
```

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 **??**.

To further compare for- and while-loops, consider the following problem.

Task 9.1

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 **??**. Often the sequence is extended with a preceding number 0, to be $0, 1, 1, 2, 3, \ldots,$

Listing 9.7 countLightweightBackwards.fsx:
Counting from 10 to 1 using a for-downto-loop using the lightweight syntax.

```
1  for i = 10 downto 1 do
2    printf "%d " i
3  printfn ""
```

```
1  $ fsharpc --nologo countLightweightBackwards.fsx && mono
     countLightweightBackwards.exe
2  10 9 8 7 6 5 4 3 2 1
```
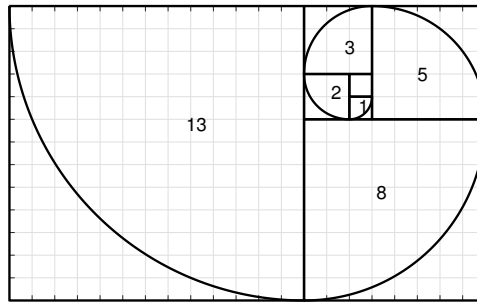

Figure 9.1: The Fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive Fibonacci numbers, and the curve in each square is the circular arc with a radius of the square it is drawn in.

which we will do here as well.

We could solve this problem with a for-loop, as shown in **??**. 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 **??** 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 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.

?? shows a program similar to **??** using a while-loop instead of for-loop. The

---

Listing 9.8 fibFor.fsx:
The n'th Fibonacci number calculated using a for-loop.

```
1  let fib n =
2    let mutable pair = (0, 1)
3    for i = 2 to n do
4      pair <- (snd pair, (fst pair) + (snd pair))
5    snd pair
6
7  printfn "fib(1) = %d" (fib 1)
8  printfn "fib(2) = %d" (fib 2)
9  printfn "fib(3) = %d" (fib 3)
10 printfn "fib(10) = %d" (fib 10)
```

```
1  $ fsharpc --nologo fibFor.fsx && mono fibFor.exe
2  fib(1) = 1
3  fib(2) = 1
4  fib(3) = 2
5  fib(10) = 55
```

---

programs are almost identical. In this case, the `for`-loop is to 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.

The correctness of `fib` in **??** can be proven using a *loop invariant*. An *invariant* is a statement that is always true at a particular point in a program, and a loop invariant is a statement which is true at the beginning and end of a loop. In line **??** in **??**, we may state the invariant: The variable `pair` is the pair of the $i - 1$'th and i'th Fibonacci numbers. This is provable by induction:

Base case: Before entering the while loop, `i` is 1, `pair` is $(0, 1)$. Thus, the invariant is true.

Induction step: Assuming that `pair` is the $i - 1$'th and i'th Fibonacci numbers, the body first assigns a new value to `pair` as the i'th and $i+1$'th Fibonacci numbers, then increases i by one such that at the end of the loop the `pair` again contains the the $i - 1$'th and i'th Fibonacci numbers.

Thus, since our invariant is true for the first case, and any iteration following an iteration where the invariant is true, is also true, then it is true for all iterations.

Thus we know that the second value in `pair` holds the value of the i'th Fibonacci number, and since we further may prove that $i = n$ when line **??** is reached, then it is proven that `fib` returns the n'th Fibonacci number.

> **Listing 9.9 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)
> ```
> ```
> $ fsharpc --nologo fibWhile.fsx && mono fibWhile.exe
> fib(1) = 1
> fib(2) = 1
> fib(3) = 2
> fib(10) = 55
> ```

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 **??**. The strategy here is to iteratively calculate Fibonacci 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.

9.2 Conditional Expressions

Programs often contain code which should only be executed under certain conditions. This can be expressed with `if`-expressions, whose syntax is as follows.

> **Listing 9.11: 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

> **Listing 9.10 fibWhileLargest.fsx:**
> **Search for the largest Fibonacci number less than a specified number.**
>
> ```
> 1  let largestFibLeq n =
> 2    let mutable pair = (0, 1)
> 3    while snd pair <= n do
> 4      pair <- (snd pair, fst pair + snd pair)
> 5    fst pair
> 6
> 7  for i = 1 to 10 do
> 8    printfn "largestFibLeq(%d) = %d" i (largestFibLeq i)
> ```
> -
> ```
> 1  $ fsharpc --nologo fibWhileLargest.fsx && mono
>      fibWhileLargest.exe
> 2  largestFibLeq(1) = 1
> 3  largestFibLeq(2) = 2
> 4  largestFibLeq(3) = 3
> 5  largestFibLeq(4) = 3
> 6  largestFibLeq(5) = 5
> 7  largestFibLeq(6) = 5
> 8  largestFibLeq(7) = 5
> 9  largestFibLeq(8) = 8
> 10 largestFibLeq(9) = 8
> 11 largestFibLeq(10) = 8
> ```

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 evaluate. If all conditions are false then the `else`-branch is evaluated. If no `else` expression is present, then "()" will be returned. See **??** for a simple example. The lightweight syntax allows for newlines entered everywhere, but indentation must be used to express scope.

To demonstrate conditional expressions, let us write a program which writes the sentence "I have n apple(s)", where the plural 's' is added appropriately for various n's. This is done in **??**, using the lightweight syntax. The sentence structure and its variants give rise to a more compact solution, since the language to be returned to the user is a variant of "I have/owe no/number apple(s)", i.e., certain conditions determine whether the sentence should use "have" and "owe" and so forth. So, we could instead make decisions on each of these sentence parts, and then built the final sentence from its parts. This is accomplished in the following example: While arguably shorter, this solution is also denser, and most likely more difficult to debug and maintain.

> **Listing 9.12 condition.fsx:**
> **Conditions evaluate their branches depending on the value of the condition.**
>
> ```
> 1  if true then printfn "hi" else printfn "bye"
> 2  if false then printfn "hi" else printfn "bye"
> ```
> --
> ```
> 1  $ fsharpc --nologo condition.fsx && mono condition.exe
> 2  hi
> 3  bye
> ```

Note that both `elif` and `else` branches are optional, which may cause problems. For example, both

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

and

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

are invalid, since F# is not smart enough to realize that the type of the expression is uniquely determined. Instead, F# looks for the `else` to ensure all cases have been covered, and that `a` always will be given a unique value of the same type regardless of the branch taken in the conditional statement. 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.

Listing 9.13 conditionalLightweight.fsx:
Using conditional expression to generate different strings.

```
1   let applesIHave n =
2     if n < -1 then
3       "I owe " + (string -n) + " apples"
4     elif n < 0 then
5       "I owe " + (string -n) + " apple"
6     elif n < 1 then
7       "I have no apples"
8     elif n < 2 then
9       "I have 1 apple"
10    else
11      "I have " + (string n) + " apples"
12
13  printfn "%A" (applesIHave -3)
14  printfn "%A" (applesIHave -1)
15  printfn "%A" (applesIHave 0)
16  printfn "%A" (applesIHave 1)
17  printfn "%A" (applesIHave 2)
18  printfn "%A" (applesIHave 10)
```

```
1   $ fsharpc --nologo conditionalLightWeight.fsx && mono
      conditionalLightWeight.exe
2   "I owe 3 apples"
3   "I owe 1 apple"
4   "I have no apples"
5   "I have 1 apple"
6   "I have 2 apples"
7   "I have 10 apples"
```

> **Listing 9.14 conditionalLightweightAlt.fsx:**
> **Using sentence parts to construct the final sentence.**

```
1  let applesIHave n =
2    let haveOrOwe = if n < 0 then "owe" else "have"
3    let pluralS = if (n = 0) || (abs n) > 1 then "s" else ""
4    let number = if n = 0 then "no" else (string (abs n))
5
6    "I " + haveOrOwe + " " + number + " apple" + pluralS
7
8  printfn "%A" (applesIHave -3)
9  printfn "%A" (applesIHave -1)
10 printfn "%A" (applesIHave 0)
11 printfn "%A" (applesIHave 1)
12 printfn "%A" (applesIHave 2)
13 printfn "%A" (applesIHave 10)
```

```
1  $ fsharpc --nologo conditionalLightWeightAlt.fsx && mono
     conditionalLightWeightAlt.exe
2  "I owe 3 apples"
3  "I owe 1 apple"
4  "I have no apples"
5  "I have 1 apple"
6  "I have 2 apples"
7  "I have 10 apples"
```

9.3 Programming Intermezzo: Automatic Conversion of Decimal to Binary Numbers

Using loops and conditional expressions, we are now able to solve the following problem:

> **Task 9.2**
>
> Given an integer on decimal form, write its equivalent value on the binary form.

To solve this problem, consider odd numbers: They all have the property that the least significant bit is 1, e.g., $1_2 = 1, 101_2 = 5$, in contrast to even numbers such as $110_2 = 6$. Division by 2 is equal to right-shifting by 1, e.g., $1_2/2 = 0.1_2 = 0.5, 101_2/2 = 10.1_2 = 2.5, 110_2/2 = 11_2 = 3$. Thus, through dividing by 2 and checking the remainder, we may sequentially read off the least significant bit. This leads to the algorithm shown in **??**. In the code, the states v and str are iteratively updated until str finally contains the desired solution.

To prove that **??** calculates the correct sequence, we use induction. First we realize that for $v < 1$, the while-loop is skipped, and the result is trivially true. We will concentrate on line **??** in **??** and will prove the following loop invariant: The string str contains all the bits of n to the right of the bit pattern remaining in variable v.

Base case $n = 000\ldots000x$: If n only uses the lowest bit, then $n = 0$ or $n = 1$. If $n = 0$, then it is trivially correct. Considering the case $n = 1$: Before entering into the loop, v is 1, and str is the empty string, so the invariant is true. The condition of the while-loop is $1 > 0$, so execution enters the loop. Since integer division of 1 by 2 gives 0 with remainder 1, str is set to `"1"` and v to 0. Now we reexamine the while-loop's condition, $0 > 0$, which is false, so we exit the loop. At this point, v is 0 and str is `"1"`, so all bits have been shifted from n to str, and none are left in v. Thus the invariant is true. Finally, the program returns `"0b1"`.

Induction step: Consider the case of $n > 1$, and assume that the invariant is true when entering the loop, i.e., that m bits already have been shifted to str and that $n > 2^m$. In this case, v contains the remaining bits of n, which is the integer division v = n / 2**m. Since $n > 2^m$, v is non-zero, and the loop conditions is true, so we enter the loop body. In the loop body we concatenate the rightmost bit of v to the left of str using v % 2, and right-shift v one bit to the right with v <- v / 2. Thus, when returning to the condition the invariant is true, since the right-most bit in v has been shifted to str. This continues until all bits have been shifted to str and v = 0, in which case the loop terminates, and `"0b"`+str is returned.

Listing 9.15 dec2bin.fsx:
Using integer division and remainder to write any positive integer in binary form.

```
1  let dec2bin n =
2    if n < 0 then
3      "Illegal value"
4    elif n = 0 then
5      "0b0"
6    else
7      let mutable v = n
8      let mutable str = ""
9      while v > 0 do
10       str <- (string (v % 2)) + str
11       v <- v / 2
12     "0b" + str
13
14
15 printfn "%4d -> %s" -1 (dec2bin -1)
16 printfn "%4d -> %s" 0 (dec2bin 0)
17 for i = 0 to 3 do
18   printfn "%4d -> %s" (pown 10 i) (dec2bin (pown 10 i))
```

```
1  $ fsharpc --nologo dec2bin.fsx && mono dec2bin.exe
2    -1 -> Illegal value
3     0 -> 0b0
4     1 -> 0b1
5    10 -> 0b1010
6   100 -> 0b1100100
7  1000 -> 0b1111101000
```

Thus we have proven that `dec2bin` correctly converts integers to strings representing binary numbers.

10 Organising Code in Libraries and Application Programs

In this chapter, we will focus on a number of ways to make the code available as *library* functions in F#. A library is a collection of types, values, and functions that an application program can use. A library does not perform calculations on its own.

F# includes several programming structures to organize code in libraries: Modules, namespaces, and classes. In this chapter, we will describe modules and namespaces. Classes will be described in detail in **??**.

10.1 Modules

An F# *module*, not to be confused with a Common Language Infrastructure module (see **??**), is a programming structure used to organize type declarations, values, functions, etc.

Every implementation and script file in F# implicitly defines a module, and the module name is given by the filename. Consider the script file `Meta.fsx` shown in **??**. Here, we have implicitly defined a module with the name `Meta`. Another script file

Listing 10.1 Meta.fsx:
A script file defining the `apply` function.

```
1  type floatFunction = float -> float -> float
2  let apply (f : floatFunction) (x : float) (y : float) : float = f
     x y
```

may now use this function, which is accessed using the "`.`" notation, i.e., `Meta.apply` will refer to this function in other programs. An application program could be as the one shown in **??**. In the example above, we have explicitly used the module's type definition for illustration purposes. A shorter and possibly simpler program would have been to define `add` as `let add x y = x + y`, since F#'s type system will infer the implied type. However, **explicit definitions of types is recommended**

> **Listing 10.2 MetaApp.fsx:**
> **Defining a script calling the module.**
>
> ```
> 1  let add : Meta.floatFunction = fun x y -> x + y
> 2  let result = Meta.apply add 3.0 4.0
> 3  printfn "3.0 + 4.0 = %A" result
> ```

for readability. Hence, an alternative to the above example's use of anonymous functions is: `let` `add (x: float) (y: float) : float = x + y`. To compile the module and the application program, we write as demonstrated in **??**. Since the

> **Listing 10.3: Compiling both the module and the application code.**
> **Note that file order matters when compiling several files.**
>
> ```
> 1  $ fsharpc --nologo Meta.fsx MetaApp.fsx && mono MetaApp.exe
> 2  3.0 + 4.0 = 7.0
> ```

F# compiler reads through the files once, the order of the filenames in the compile command is very important. Hence, the script containing the module and function definitions must be to the left of the script containing their use. Notice also that if not otherwise specified, the F# compiler produces an `.exe` file derived from the last filename in the list of filenames.

We may also explicitly define the module name using the *module* with the following syntax,

> **Listing 10.4: Outer module.**
>
> ```
> 1  module <ident>
> 2  <script>
> ```

Here, the identifier `<ident>` is a name not necessarily related to the filename, and the script `<script>` is an expression. An example is given in **??**. Since we have created a new file, where the module `Meta` is explicitly defined, we can use the same application program. This is demonstrated in **??**. Since `MetaExplicit.fsx` explicitly defines the module name, `apply` is not available to an application program as `MetaExplicit.apply`. **It is recommended that module names are defined explicitly, since filenames may change due to external conditions.** In other words, filenames are typically set from the perspective of the filesystem. The user may choose to change names to suit a filesystem structure, or different platforms may impose different file naming conventions. Thus, direct linking of filenames with the

Listing 10.5 MetaExplicit.fsx:
Explicit definition of the outermost module.

```
1  module Meta
2  type floatFunction = float -> float -> float
3  let apply (f : floatFunction) (x : float) (y : float) : float = f
     x y
```

Listing 10.6: Changing the module definition to explicit naming has
no effect on the application nor the compile command.

```
1  $ fsharpc --nologo MetaExplicit.fsx MetaApp.fsx && mono
     MetaApp.exe
2  3.0 + 4.0 = 7.0
```

internal workings of a program is a needless complication of structure.

The definitions inside a module may be accessed directly from an application program, omitting the ".”-notation, by use of the *open* keyword,

Listing 10.7: Open module.

```
1  open <ident>
```

We can modify `MetaApp.fsx`, as shown in **??**.   In this case, the namespace of our

Listing 10.8 MetaAppWOpen.fsx:
Avoiding the ".”-notation by the open keyword.

```
1  open Meta
2  let add : floatFunction = fun x y -> x + y
3  let result = apply add 3.0 4.0
4  printfn "3.0 + 4.0 = %A" result
```

previously defined module is included into the scope of the application functions, and its types, values, functions, etc. can be used directly, as shown in **??**.   The open-keyword should be used sparingly, since including a library's definitions into the application scope can cause surprising naming conflicts, because the user of a library typically has no knowledge of the inner workings of the library. E.g., the user may accidentally use code defined in the library, but with different type and functionality than intended, which the type system will use to deduce types in the

Listing 10.9: How the application program opens the module has no effect on the module code nor compile command.

```
1  $ fsharpc --nologo MetaExplicit.fsx MetaAppWOpen.fsx &&
     mono MetaAppWOpen.exe
2  3.0 + 4.0 = 7.0
```

application program, and therefore will either give syntax or runtime errors that are difficult to understand. This problem is known as *namespace pollution*, and for clarity, **it is recommended to use the open-keyword sparingly**. Note that for historical reasons, the phrase 'namespace pollution' is used to cover pollution both due to modules and namespaces.

Modules may also be nested, in which case the nested definitions must use the "="-sign and must be appropriately indented.

Listing 10.10:   Nested modules.

```
1  module <ident> = <script>
```

In lightweight syntax, a newline may be entered before the script `<script>`, and the script must be indented. An example is shown in **??**. In this case, `Meta` and `MathFcts`

Listing 10.11 nestedModules.fsx:
Modules may be nested.

```
1  module Utilities
2  let PI = 3.1415
3  module Meta =
4    type floatFunction = float -> float -> float
5    let apply (f : floatFunction) (x : float) (y : float) : float =
     f x y
6  module MathFcts =
7    let add : Meta.floatFunction = fun x y -> x + y
```

are defined at the same level and said to be siblings, while `Utilities` is defined at a higher level. In this relation, the former two are said to be the children of the latter. Note that the nesting respects the lexical scope rules, such that the constant `PI` is directly accessible in both modules `Meta` and `MathFcts`, as is the module `Meta` in `MathFcts`, but not `MathFcts` in `Meta`. The "."-notation is reused to index deeper into the module hierarchy, as the example in **??** shows.   Modules can be recursive using the *rec*-keyword, meaning that in our example we can make the outer module

Listing 10.12 nestedModulesApp.fsx:
Applications using nested modules require additional usage of the " ."
notation to navigate the nesting tree.

```
1
2  let add : Utilities.Meta.floatFunction = fun x y -> x + y
3  let result = Utilities.Meta.apply Utilities.MathFcts.add 3.0
      Utilities.PI
4  printfn "3.0 + 4.0 = %A" result
```

recursive, as demonstrated in **??**. The consequence is that the modules `Meta` and

Listing 10.13 nestedRecModules.fsx:
Mutual dependence on nested modules requires the `rec` keyword in
the module definition.

```
1  module rec Utilities
2    module Meta =
3      type floatFunction = float -> float -> float
4      let apply (f : floatFunction) (x : float) (y : float) : float
    = f x y
5    module MathFcts =
6      let add : Meta.floatFunction = fun x y -> x + y
```

`MathFcts` are accessible in both modules, but compilation will now give a warning
since soundness of the code will first be checked at runtime. In general, it is advised
to **avoid programming constructions whose validity cannot be checked at**
compile-time.

10.2 Namespaces

An alternative way to structure code in modules is to use a *namespace*, which can only
hold modules and type declarations and only works in compiled mode. Namespaces
are defined as explicitly defined outer modules, using the *namespace* keyword in
accordance with the following syntax.

Listing 10.14:   Namespace.

```
1  namespace <ident>
2  <script>
```

An example is given in **??**.   Notice that when organizing code in a namespace, the

---

Listing 10.15 namespace.fsx:
Defining a namespace is similar to explicitly named modules.

```
1  namespace Utilities
2  type floatFunction = float -> float -> float
3  module Meta =
4    let apply (f : floatFunction) (x : float) (y : float) : float =
     f x y
```

---

first line of the file, other than comments and compiler directives, must be the one
starting with `namespace`.

As for modules, the content of a namespace is accessed using the "`.`" notation, as
demonstrated in **??**.   Likewise, the compilation is performed in the same way as for

---

Listing 10.16 namespaceApp.fsx:
**The "`.`"-notation lets the application program access functions and
types in a namespace.**

```
1  let add : Utilities.floatFunction = fun x y -> x + y
2  let result = Utilities.Meta.apply add 3.0 4.0
3  printfn "3.0 + 4.0 = %A" result
```

---

modules, see **??**.   Hence, from an application point of view, it is not immediately

---

**Listing 10.17: Compilation of files including namespace definitions uses
the same procedure as modules.**

```
1  $ fsharpc --nologo namespace.fsx namespaceApp.fsx && mono
   namespaceApp.exe
2  3.0 + 4.0 = 7.0
```

---

possible to see that `Utilities` is defined as a namespace and not a module. However,
in contrast to modules, namespaces may span several files. E.g., we may add a third
file extending the `Utilities` namespace with the `MathFcts` module, as demonstrated
in **??**.   To compile, we now need to include all three files in the right order, see **??**.
 The order matters:`namespaceExtension.fsx` uses the definition of `floatFunction`
in the file `namespace.fsx`. You can use extensions to extend existing namespaces
included with the F# compiler.

Namespaces may also be nested. In contrast to modules, nesting is defined using the

Listing 10.18 namespaceExtension.fsx:
Namespaces may span several files. Here is shown an extra file which
extends the `Utilities` namespace.

```
1  namespace Utilities
2  module MathFcts =
3    let add : floatFunction = fun x y -> x + y
```

Listing 10.19: Compilation of namespaces defined in several files re-
quires careful consideration of order, since the compiler reads once
and only once through the files in the order they are given.

```
1  $ fsharpc --nologo namespace.fsx namespaceExtension.fsx
     namespaceApp.fsx && mono namespaceApp.exe
2  3.0 + 4.0 = 7.0
```

"." notation. That is, to create a child namespace `more` of `Utilities`, we must use
initially write `namespace Utilities.more`. Indentation is ignored in the `namespace`
line, thus left-most indentation is almost always used. Namespaces follow lexical
scope rules, and identically to modules, namespaces containing mutually dependent
children can be declared using the `rec` keyword, e.g., `namespace rec Utilities`.

10.3 Compiled Libraries

Libraries may be distributed in compiled form as `.dll` files. This saves the user
from having to recompile a possibly large library every time library functions needs
to be compiled with an application program. In order to produce a library file from
`MetaExplicit.fsx` and then compile an application program, we first use the com-
piler's `-a` option to produce the `.dll`. A demonstration is given in **??**. This produces

Listing 10.20: A stand-alone `.dll` file is created and used with special
compile commands.

```
1  $ fsharpc --nologo -a MetaExplicit.fsx
```

the file `MetaExplicit.dll`, which may be linked to an application by using the `-r`
option during compilation, see **??**. A library can be the result of compiling a number
of files into a single `.dll` file. `.dll`-files may be loaded dynamically in script files
(`.fsx`-files) by using the *#r directive*, as illustrated in **??**. We may now omit the

---

Listing 10.21: The library is linked to an application during compilation to produce runnable code.

```
1  $ fsharpc --nologo -r MetaExplicit.dll MetaApp.fsx && mono
     MetaApp.exe
2  3.0 + 4.0 = 7.0
```

Listing 10.22 MetaHashApp.fsx:
The `.dll` file may be loaded dynamically in `.fsx` script files and in interactive mode. Nevertheless, this usage is not recommended.

```
1  #r "MetaExplicit.dll"
2  let add : Meta.floatFunction = fun x y -> x + y
3  let result = Meta.apply add 3.0 4.0
4  printfn "3.0 + 4.0 = %A" result
```

---

explicit mentioning of the library when compiling, as shown in **??**. The `#r` directive

---

Listing 10.23: When using the #r directive, then the `.dll` file need not be explicitly included in the list of files to be compiled.

```
1  $ fsharpc --nologo MetaHashApp.fsx && mono MetaHashApp.exe
2  3.0 + 4.0 = 7.0
```

---

is also used to include a library in interactive mode. However, for the code to be compiled, the use of the `#r` directive requires that the filesystem path to the library is coded inside the script. As for module names, direct linking of filenames with the internal workings of a program is a needless complication of structure, and **it is recommended not to rely on the use of the #r directive.**

In the above listings we have compiled *script files* into libraries. However, F# has reserved the `.fs` filename suffix for library files, and such files are called *implementation files*. In contrast to script files, implementation files do not support the `#r` directive. When compiling a list of implementation and script files, all but the last file must explicitly define a module or a namespace.

Both script and implementation files may be augmented with *signature files*. A signature file contains no implementation, 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 **??**.

Signature files can be generated automatically using the `--sig:<filename>` compiler directive. To demonstrate this feature, we will first move the definition of `add` to the implementation file, see **??**.   A signature file may be automatically generated, as

---

Listing 10.24 MetaWAdd.fs:
An implementation file including the `add` function.

```
1  module Meta
2  type floatFunction = float -> float -> float
3  let apply (f : floatFunction) (x : float) (y : float) : float = f
      x y
4  let add (x : float) (y : float) : float = x + y
```

---

shown in **??**.   The warning can safely be ignored, since at this point it is not our

---

Listing 10.25: Automatic generation of a signature file at compile time.

```
1  $ fsharpc --nologo --sig:MetaWAdd.fsi -a MetaWAdd.fs
```

---

intention to produce runnable code. The above listing has generated the signature file in **??**.   We can generate a library using the automatically generated signature file by

---

Listing 10.26 MetaWAdd.fsi:
An automatically generated signature file from `MetaWAdd.fs`.

```
1  module Meta
2  type floatFunction = float -> float -> float
3  val apply : f:floatFunction -> x:float -> y:float -> float
4  val add : x:float -> y:float -> float
```

---

writing `fsharpc -a MetaWAdd.fsi MetaWAdd.fs`, which is identical to compiling the

`.dll` file without the signature file. However, if we remove, e.g., the type definition for `add` in the signature file, then this function becomes private to the module and cannot be accessed outside. Hence, using the signature file in **??**, and recompiling the `.dll` with **??** does not generates errors.   However, when using the newly created

---

Listing 10.27 MetaWAddRemoved.fsi:
Removing the type defintion for `add` from `MetaWAdd.fsi`.

```
1  module Meta
2  type floatFunction = float -> float -> float
3  val apply : f:floatFunction -> x:float -> y:float -> float
```

Listing 10.28: Automatic generation of a signature file at compile time.

```
1  $ fsharpc --nologo -a MetaWAdd.fsi MetaWAdd.fs
```

`MetaWAdd.dll` with an application that does not itself supply a definition of `add`, we get a syntax error, since `add` now is inaccessible to the application program. This is demonstrated in **????**.

Listing 10.29 MetaWOAddApp.fsx:
A version of ?? without a definition of `add`.

```
1  let result = Meta.apply add 3.0 4.0
2  printfn "3.0 + 4.0 = %A" result
```

Listing 10.30: Automatic generation of a signature file at compile time.

```
1  $ fsharpc --nologo -r MetaWAdd.dll MetaWAddRemoved.fsi
      MetaWOAddApp.fsx
2
3  MetaWOAddApp.fsx(1,25): error FS0039: The value or
      constructor 'add' is not defined.
4
5  MetaWAddRemoved.fsi(1,1): error FS0240: The signature file
      'Meta' does not have a corresponding implementation
      file. If an implementation file exists then check the
      'module' and 'namespace' declarations in the signature
      and implementation files match.
```

11 Testing Programs

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 **??**. Software is everywhere, and errors therein have a
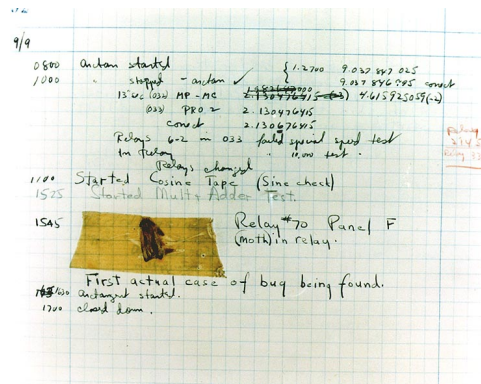
Figure 11.1: The first computer bug, caught by Grace Hopper, U.S. Naval Historical Center Online Library Photograph NH 96566-KN.

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?

---

[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

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?

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:

> **Task 11.1**
>
> Given 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 **??**.

11.1 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

Listing 11.1 date2Day.fsx:
A function that can calculate day-of-week from any date in the Gregorian calendar.

```
1  let januaryFirstDay (y : int) =
2    let a = (y - 1) % 4
3    let b = (y - 1) % 100
4    let c = (y - 1) % 400
5    (1 + 5 * a + 4 * b + 6 * c) % 7
6
7  let rec sum (lst : int list) j =
8    if 0 <= j && j < lst.Length then
9      lst.[0] + sum lst.[1..] (j - 1)
10   else
11     0
12
13 let date2Day d m y =
14   let dayPrefix =
15     ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri"; "Satur"]
16   let feb = if (y % 4 = 0) && ((y % 100 <> 0) || (y % 400 = 0))
17    then 29 else 28
17   let daysInMonth = [31; feb; 31; 30; 31; 30; 31; 31; 30; 31; 30;
     31]
18   let dayOne = januaryFirstDay y
19   let daysSince = (sum daysInMonth (m - 2)) + d - 1
20   let weekday = (dayOne + daysSince) % 7;
21   dayPrefix.[weekday] + "day"
```

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 **??** has 3 functions, and we will consider these each as a unit, but we might as well just have chosen 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. In this example, there are only examples of if-branch points,

but they may as well be loops and pattern matching expressions. In the **??** it is shown that the branch points have been given a comment and a number.

Listing 11.2 date2DayAnnotated.fsx:
In white-box testing, the branch points are identified.

```
1   // Unit: januaryFirstDay
2   let januaryFirstDay (y : int) =
3     let a = (y - 1) % 4
4     let b = (y - 1) % 100
5     let c = (y - 1) % 400
6     (1 + 5 * a + 4 * b + 6 * c) % 7
7
8   // Unit: sum
9   let rec sum (lst : int list) j =
10    (* WB: 1 *)
11    if 0 <= j && j < lst.Length then
12      lst.[0] + sum lst.[1..] (j - 1)
13    else
14      0
15
16  // Unit: date2Day
17  let date2Day d m y =
18    let dayPrefix =
19      ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri"; "Satur"]
20    (* WB: 1 *)
21    let feb = if (y % 4 = 0) && ((y % 100 <> 0) || (y % 400 =
       0)) then 29 else 28
22    let daysInMonth = [31; feb; 31; 30; 31; 30; 31; 31; 30;
       31; 30; 31]
23    let dayOne = januaryFirstDay y
24    let daysSince = (sum daysInMonth (m - 2)) + d - 1
25    let weekday = (dayOne + daysSince) % 7;
26    dayPrefix.[weekday] + "day"
```

---

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 **??**. 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 **??**.

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

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 11.1: Unit test

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.

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.

11.2 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

Listing 11.3 date2DayWhiteTest.fsx:
The tests identified by white-box analysis. The program from ??
has been omitted for brevity.

```
1  printfn "White-box testing of date2Day.fsx"
2  printfn "  Unit: januaryFirstDay"
3  printfn "    Branch: 0 - %b" (januaryFirstDay 2016 = 5)
4
5  printfn "  Unit: sum"
6  printfn "    Branch: 1a - %b" (sum [1; 2; 3] 1 = 3)
7  printfn "    Branch: 1b - %b" (sum [1; 2; 3] -1 = 0)
8  printfn "    Branch: 1c - %b" (sum [1; 2; 3] 10 = 0)
9
10 printfn "  Unit: date2Day"
11 printfn "    Branch: 1a - %b" (date2Day 8 9 2016 =
      "Thursday")
12 printfn "    Branch: 1b - %b" (date2Day 8 9 2000 = "Friday")
13 printfn "    Branch: 1c - %b" (date2Day 8 9 2100 =
      "Wednesday")
14 printfn "    Branch: 1d - %b" (date2Day 8 9 2015 = "Tuesday")
```

```
1  $ fsharpc --nologo date2DayWhiteTest.fsx && mono
      date2DayWhiteTest.exe
2  White-box testing of date2Day.fsx
3    Unit: januaryFirstDay
4      Branch: 0 - true
5    Unit: sum
6      Branch: 1a - true
7      Branch: 1b - true
8      Branch: 1c - true
9    Unit: date2Day
10     Branch: 1a - true
11     Branch: 1b - true
12     Branch: 1c - true
13     Branch: 1d - true
```

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:

    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 **??**.

4. Write a program executing the tests, as shown in **????**. Notice how the program has been made such that it is almost a direct copy of the table produced in the previous step.

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.

11.3 Debugging by Tracing

Once an error has been found by testing, the *debugging* phase starts. The cause of a bug can either be that the chosen algorithm is the wrong one for the job, or

Test number	Input	Expected output
??a	1 1 2016	Friday
??b	2 1 2016	Saturday
??c	3 1 2016	Sunday
??d	4 1 2016	Monday
??e	5 1 2016	Tuesday
??f	6 1 2016	Wednesday
??g	7 1 2016	Thursday
??a	31 12 2014	Wednesday
??b	1 1 2015	Thursday
??c	30 9 2017	Saturday
??d	1 10 2017	Sunday
??a	28 2 2016	Sunday
??b	29 2 2016	Monday
??c	1 3 2016	Tuesday
??d	28 2 2017	Tuesday
??e	1 3 2017	Wednesday
??a	1 3 2015	Sunday
??b	1 3 2012	Thursday
??c	1 3 2000	Wednesday
??d	1 3 2100	Monday

Table 11.2: Black-box testing

Listing 11.4 date2DayBlackTest.fsx:
The tests identified by black-box analysis. The program from ??
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)
```

Listing 11.5: Output from ??.

```
1  $ fsharpc --nologo date2DayBlackTest.fsx && mono
      date2DayBlackTest.exe
2  Black-box testing of date2Day.fsx
3    1. A complete week
4       test 1 - true
5       test 2 - true
6       test 3 - true
7       test 4 - true
8       test 5 - true
9       test 6 - true
10      test 7 - true
11   2. Across boundaries
12      test 1 - true
13      test 2 - true
14      test 3 - true
15      test 4 - true
16   3. Across Feburary boundary
17      test 1 - true
18      test 2 - true
19      test 3 - true
20      test 4 - true
21      test 5 - true
22   4. Leap years
23      test 1 - true
24      test 2 - true
25      test 3 - true
26      test 4 - true
```

the implementation of it has an error. In the debugging process, we have to keep an open mind and not rely on assumptions. A frequent source of errors is that the state of a program is different than expected, e.g., because the calculation performed is different than intended, or that the return of a library function is different than expected. The most important tool for debugging is *simplification*. This is similar to white-box testing, but where the units tested are very small. E.g., the suspected piece of code could be broken down into smaller functions or code snippets, which are given well-defined input, and, e.g., use `printfn` statements to obtain the output of the code snippet. Another related technique is to use *mockup code*, that is, replacing parts of the code with code that produces safe and relevant results. If the bug is not obvious, then more rigorous techniques must be used, such as *tracing*. Some development interfaces have a built-in tracing system, e.g., `fsharpi` will print inferred types and some binding values. However, often the source of a bug is due to a misunderstanding of the flow of data through a program execution, and we will in the following section introduce *Trace by hand* as a technique to simulate the execution of a program by hand. In the following section, tracing will refer to the Trace by hand method.

To understand the method of Tracing by hand, we will consider 3 imperative programs of gradually increasing complexity: a program using function call, a program including a `for`-loop, and a program with dynamic scope. In **??** we give a fourth example using recursion, a concept to be introduced in **??**.

Tracing may seem tedious in the beginning, but in conjunction with strategically placed debugging `printfn` statements, it is a very valuable tool for debugging.

11.3.1 Tracing Function Calls

Consider the program in **??**. The program calls `testScope 2.0`, and by running

Listing 11.6 lexicalScopeTracing.fsx:
Example of lexical scope and closure environment.

```
1  let testScope x =
2    let a = 3.0
3    let f z = a * z
4    let a = 4.0
5    f x
6  printfn "%A" (testScope 2.0)
```

```
1  $ fsharpc --nologo lexicalScopeTracing.fsx && mono
     lexicalScopeTracing.exe
2  6.0
```

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 **??** contains a function definition and a call, hence, the first lines of our table looks like,

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

The elements of the table is 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 **??**, hence, in Step 1, we update the enviroment with the binding of a function to the name `testScope`. Since functions are values in F#, we note their bindings by their closures: a tuple of argument names, the function-body, and the values lexically available at the place of binding. See **??** for more information on closures. Following the function-binding, the `printfn` statement is called in line **??** 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	??	E_1	$\big((x = 2.0), \text{testScope-body}, ()\big)$

This means that we are going to execute the code in testScope-body. The function was called with 2.0 as 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	??	E_1	$a = 3.0$
5	??	E_1	$f = \big((z), \text{a * z}, (a = 3.0, x = 2.0)\big)$
6	??	E_1	$a = 4.0$
7	??	E_1	$\text{f } x = ?$

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 **??**, 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	??	E_2	$\big((z = 2.0), \text{a * z}, (a = 3.0, x = 2.0)\big)$

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	??	E_2	$a * z = 6.0$
10	??	E_2	$\text{return} = 6.0$

The 'return'-word is used to remind us that this is the value to replace the question mark with in 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	??	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	??	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	??	E_0	return = ()

The full table is shown for completeness in **??**. Hence, we conclude that the program

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

Table 11.3: The complete table produced while tracing the program in **??** by hand.

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 11.7 lexicalScopeTracingCorrected.fsx:
Tracing the code in ?? by hand produced the table in ??, and to get
the desired output, we correct the code as shown here.

```
1  let testScope x =
2    let a = 4.0
3    let f z = a * z
4    f x
5  printfn "%A" (testScope 2.0)
```

```
1  $ fsharpc --nologo lexicalScopeTracingCorrected.fsx &&
     mono lexicalScopeTracingCorrected.exe
2  8.0
```

11.3.2 Tracing Loops

Consider the program in **??**. The program includes a function for printing the

Listing 11.8 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
```

```
1  $ fsharpc --nologo printSquares.fsx && mono
     printSquares.exe
2  1: 1
3  2: 4
4  3: 9
```

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 **??**. The unfolding contains
3 new scopes lines **??**–**??**, lines **??**–**??**, and lines **??**–**??** corresponding to the 3 times,
the loop is repeated, and each scope starts by binding the counting value to the name
`i`.

Listing 11.9 printSquaresUnfold.fsx:
An unfolded version of ??.

```
1   let N = 3
2   let printSquaresUnfold n =
3     (
4       let i = 1
5       let p = i * i
6       printfn "%d: %d" i p
7     )
8     (
9       let i = 2
10      let p = i * i
11      printfn "%d: %d" i p
12    )
13    (
14      let i = 3
15      let p = i * i
16      printfn "%d: %d" i p
17    )
18
19  printSquaresUnfold N
```

```
1   $ fsharpc --nologo printSquaresUnfold.fsx && mono
       printSquaresUnfold.exe
2   1: 1
3   2: 4
4   3: 9
```

In the rest of this section, we will refer to the code in **??**. The first rows in our tracing-table looks as follows:

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	??	E_0	$N = 3$
2	??	E_0	printSquares $= \big((n), \text{printSquares-body}, (N = 3)\big)$
3	??	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,

Step	Line	Env.	Bindings and evaluations
4	??	E_1	$\big((n = 3), \text{printSquares-body}, (N = 3)\big)$

The first statement of printSquares-body is the `for`-loop. As our unfolding in **??**

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	??	E_1	for $\cdots = ?$

and create a new environment as if it had been a function,

Step	Line	Env.	Bindings and evaluations
6	??	E_2	$\big((i = 1), \text{for-body}, (n = 3, N = 3)\big)$

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	??	E_2	$i * i = 1$
8	??	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	??	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	??	E_3	$\big((i = 2), \text{for-body}, (n = 3, N = 3)\big)$
11	??	E_3	$i * i = 4$
12	??	E_3	$p = 4$
13	??	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	??	E_4	$\big((i = 3), \text{for-body}, (n = 3, N = 3)\big)$
15	??	E_4	$i * i = 9$
16	??	E_4	$p = 9$
17	??	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	??	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	??	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	??	E_0	return $= ()$

11.3.3 Tracing Mutable Values

For mutable bindings, the scope is dynamic. For this, we need the concept of storage. Consider the program in **??**.   To trace the dynamic behavior of this program, we

> **Listing 11.10 dynamicScopeTracing.fsx:**
> **Example of lexical scope and closure environment.**

```
1  let testScope x =
2    let mutable a = 3.0
3    let f z = a * z
4    a <- 4.0
5    f x
6  printfn "%A" (testScope 2.0)
```
```
1  $ fsharpc --nologo dynamicScopeTracing.fsx && mono
     dynamicScopeTracing.exe
2  8.0
```

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 **??**, the firsts 4 steps thus look like,

Step	Line	Env.	Bindings and evaluations	Step	Value
0	-	E_0	()	0	-
1	??	E_0	testScope $= \big((x), \text{body}, ()\big)$		
2	??	E_0	testScope 2.0 $= ?$		
3	??	E_1	$\big((x = 2.0), \text{body}, ()\big)$		

The mutable binding in line **??** 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	Step	Value
4	??	E_1	$a = \alpha_1$	4	$\alpha_1 = 3.0$

The following closure of `f` uses the reference-name instead of its value,

Step	Line	Env.	Bindings and evaluations	Step	Value
5	??	E_1	$f = \big((z), a * z, (x = 2.0, a = \alpha_1)\big)$	4	$\alpha_1 = 3.0$

In line **??**, the value in the storage is updated by the assignment operator, which we denote as,

Step	Line	Env.	Bindings and evaluations	Step	Value
6	??	E_1	$a \mathrel{<\!-} 4.0$	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	Step	Value
7	??	E_1	f $x = ?$	6	$\alpha_1 = 4.0$
8	??	E_2	$\big((z = 2.0), a * z, (x = 2.0, a = \alpha_1)\big)$		
9	??	E_2	$a * z = 8.0$		
10	??	E_2	return $= 8.0$		
10	??	E_1	return $= 8.0$		
11	??	E_0	output $=$ "8.0\n"		
12	??	E_0	return $= ()$		

For reference, the complete pair of tables is shown in **??**. By comparing this to the value-bindings in **??**, we see that the mutable values give rise to a different result due to the difference between lexical and dynamic scope.

Step	Line	Env.	Bindings and evaluations	Step	Value
0	-	E_0	()	0	-
1	??	E_0	testScope $= \big((x), \text{body}, ()\big)$	4	$\alpha_1 = 3.0$
2	??	E_0	testScope 2.0 = ?	6	$\alpha_1 = 4.0$
3	??	E_1	$\big((x = 2.0), \text{body}, ()\big)$		
4	??	E_1	$a = \alpha_1$		
5	??	E_1	$f = \big((z), a * z, (x = 2.0, a = \alpha_1)\big)$		
6	??	E_1	a <- 4.0		
7	??	E_1	f x = ?		
8	??	E_2	$\big((z = 2.0), a * z, (x = 2.0, a = \alpha_1)\big)$		
9	??	E_2	$a * z = 8.0$		
10	??	E_2	return = 8.0		
10	??	E_1	return = 8.0		
11	??	E_0	output = "8.0\n"		
12	??	E_0	return = ()		

Table 11.4: The complete table produced while tracing the program in **??** by hand.

12 Collections of Data

F# is tuned to work with collections of data, and there are several built-in types of collections with various properties making them useful for different tasks. Examples include strings, lists, and arrays. Strings were discussed in **??** and will be revisited here in more details.

The data structures discussed below all have operators, properties, methods, and modules to help you write elegant programs using them.

Properties and methods are common object-oriented terms used in conjunction with the discussed functionality. They are synonymous with values and functions and will be discussed in **??**. Properties and methods for a value or variable are called using the *dot notation*, i.e., with the "."-lexeme. For example, `"abcdefg"`.`Length` is a property and is equal to the length of the string, and `"abcdefg"`.`ToUpper()` is a method and creates a new string where all characters have been converted to upper case.

The data structures also have accompanying modules with a wealth of functions and where some are mentioned here. Further, the data structures are all implemented as classes offering even further functionality. The modules are optimized for functional programming, see **??**–**??**, while classes are designed to support object-oriented programming, see **??**–**??**.

In the following, a brief overview of many properties, methods, and functions is given by describing their name and type-definition, and by giving a short description and an example of their use. Several definitions are general and works with many different types. To describe this we will use the notation of generic types, see **??**. The name of a generic type starts with the "'" lexeme, such as `'T`. The implication of the appearance of a generic type in, e.g., a function's type-definition, is that the function may be used with any real type such as `int` or `char`. If the same generic type name is used in several places in the type-definition, then the function must use a real type consistently. For example, The `List.fromArray` function has type `arr:'T [] -> 'T list`, meaning that it takes an array of some type and returns a list of the same type.

See the F# Language Reference at `https://docs.microsoft.com/en-us/dotnet/fsharp/` for a full description of all available functionality including variants of those included here.

12.1 Strings

Strings have been discussed in **??**, the content of which will be briefly revisited here followed by a description of some of the many supporting built-in functions in F# on strings.

A *string* is a sequence of characters. Each character is represented using UTF-16, see **??** for further details on the unicode standard. The type `string` is an alias for *System.string*. String literals are delimited by double quotation marks ""'" and inside the delimiters, character escape sequences are allowed (see **??**), which are replaced by the corresponding character code. Examples are `"This is a string"`, `"\tTabulated string"`, `"A \"quoted\" string"`, and `""`. Strings may span several lines, and new lines inside strings are part of the string unless the line is ended with a backslash. Strings may be *verbatim* by preceding the string with "`@`", in which case escape sequences are not replaced, but two double quotation marks are an escape sequence which is replaced by a one double quotation mark. Examples of "`@`"-verbatim strings are:

- `@"This is a string"`,

- `@"\tNon-tabulated string"`,

- `@"A ""quoted"" string"`, and

- `@""`.

Alternatively, a verbatim string may be delimited by three double quotation marks. Examples of "`"""`"-verbatim strings are:

- `"""This is a string"""`,

- `"""\tNon-tabulated string"""`,

- `"""A "quoted" string"""`, and

- `""""""`.

Strings may be indexed using the `.[]` notation, as demonstrated in **??**.

12.1.1 String Properties and Methods

Strings have a few properties which are values attached to each string and accessed using the "`.`" notation. The only to be mentioned here is:

`IndexOf(): str:string -> int`.
> Returns the index of the first occurence of `s` or -1, if `str` does not appear in the string.

Listing 12.1: `IndexOf()`

```
> "Hello World".IndexOf("World");;
val it : int = 6
```

`Length: int`.
> Returns the length of the string.

Listing 12.2: `Length`

```
> "abcd".Length;;
val it : int = 4
```

`ToLower(): unit -> string`.
> Returns a copy of the string where each letter has been converted to lower case.

Listing 12.3: `ToLower()`

```
> "aBcD".ToLower();;
val it : string = "abcd"
```

`ToUpper(): unit -> string`.
> Returns a copy of the string where each letter has been converted to upper case.

Listing 12.4: `ToUpper()`

```
> "aBcD".ToUpper();;
val it : string = "ABCD"
```

`Trim(): unit -> string`.
> Returns a copy of the string where leading and trailing whitespaces have been removed.

```
Listing 12.5: Trim()
1  > "  Hello  World  ".Trim();;
2  val it : string = "Hello World"
```

`Split(): unit -> string [].`
    Splits a string of words separated by spaces into an array of words. See **??** for
    more information about arrays.

```
Listing 12.6: Split()
1  > "Hello World".Split();;
2  val it : string [] = [|"Hello"; "World"|]
```

12.1.2 The String Module

The `String` module offers many functions for working with strings. Some of the most
powerful ones are listed below, and they are all higher-order functions.

`String.collect: f:(char -> string) -> str:string -> string.`
    Creates a new string whose characters are the results of applying `f` to each of
    the characters of `str` and concatenating the resulting strings.

```
Listing 12.7: String.collect
1  > String.collect (fun c -> (string c) + ", ") "abc";;
2  val it : string = "a, b, c, "
```

`String.exists: f:(char -> bool) -> str:string -> bool.`
    Returns true if any character in `str` evaluates to true when using `f`.

```
Listing 12.8: String.exists
1  > String.exists (fun c -> c = 'd') "abc";;
2  val it : bool = false
```

`String.forall: f:(char -> bool) -> str:string -> bool.`
    Returns true if all characters in `str` evalutes to true when using `f`.

Listing 12.9: `String.forall`

```
1  > String.forall (fun c -> c < 'd') "abc";;
2  val it : bool = true
```

`String.init: n:int -> f:(int -> string) -> string`.
   Creates a new string with length `n` and whose characters are the result of applying `f` to each index of that string.

Listing 12.10: `String.init`

```
1  > String.init 5 (fun i -> (string i) + ", ");;
2  val it : string = "0, 1, 2, 3, 4, "
```

`String.iter: f:(char -> unit) -> str:string -> unit`.
   Applies `f` to each character in `str`.

Listing 12.11: `String.iter`

```
1  > String.iter (fun c -> printfn "%c" c) "abc";;
2  a
3  b
4  c
5  val it : unit = ()
```

`String.map: f:(char -> char) -> str:string -> string`.
   Creates a new string whose characters are the results of applying `f` to each of the characters of `str`.

Listing 12.12: `String.map`

```
1  > let toUpper c = c + char (int 'A' - int 'a')
2  - String.map toUpper "abcd";;
3  val toUpper : c:char -> char
4  val it : string = "ABCD"
```

12.2 Lists

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

---

Listing 12.13: The syntax for a list using the sequence expression.

```
[[<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 12.14: The syntax for a list using the range expressions.

```
[<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]`.

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 using the "`.[]`" notation, the lengths of lists is retrieved using the *Length* property, and we may test whether a list is empty by using the *IsEmpty* property. These features are demonstrated in **??**. F# implements lists as linked lists, as illustrated in **??**. As a consequence, indexing
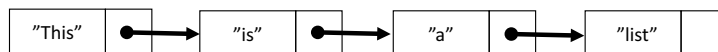
Figure 12.1: A list is a linked list: Here is illustrated the linked list of `["This"; "is"; "a"; "list"]`.

element i has *computational complexity* $\mathcal{O}(i)$. The computational complexity of an operation is a description of how long a computation will take without considering the hardware it is performed on. The notation is sometimes called *Big-O* notation or *Landau notation*. In the present case, the complexity is $\mathcal{O}(i)$, which means that the complexity is linear in i and indexing element $i+1$ takes 1 unit longer than indexing element i when i is very large. The size of the unit is on purpose unspecified and depends on implementation and hardware details. Nevertheless, Big-O notation is a useful tool for reasoning about the efficiency of an operation. F# has access to

> **Listing 12.15 listIndexing.fsx:**
> **Lists are indexed as strings and has a `Length` property.**
>
> ```
> 1  let printList (lst : int list) : unit =
> 2    for i = 0 to lst.Length - 1 do
> 3      printf "%A " lst.[i]
> 4    printfn ""
> 5
> 6  let lst = [3; 4; 5]
> 7  printfn "lst = %A, lst.[1] = %A" lst lst.[1]
> 8  printfn "lst.Length = %A, lst.isEmpty = %A" lst.Length lst.IsEmpty
> 9  printList lst
> ```
> ---
> ```
> 1  $ fsharpc --nologo listIndexing.fsx && mono
>      listIndexing.exe
> 2  lst = [3; 4; 5], lst.[1] = 4
> 3  lst.Length = 3, lst.isEmpty = false
> 4  3 4 5
> ```

the list's elements only by traversing the list from its beginning. I.e., to obtain the value of element i, F# starts with element 0, follows the link to element 1 and so on, until element i is reached. To reach element $i+1$ instead, we would need to follow 1 more link, and assuming that following a single link takes some constant amount of time we find that the computational complexity is $\mathcal{O}(i)$. Compared to arrays, to be discussed below, this is slow, which is why **indexing lists should be avoided.**

Notice especially that lists are zero-indexed, and thus, the last element in a list `lst` is `lst.Length -1`. This is a very common source of error! Therefore, indexing in lists using *for*-loops is supported using a special notation with the ***in*** keyword,

> **Listing 12.16:   For-in loop with in expression.**
>
> ```
> 1  for <ident> in <list> do <bodyExpr> [done]
> ```

In `for`-`in` loops, the loop runs through each element of the `<list>`, and assigns it to the identifier `<ident>`. This is demonstrated in **??**.   Using `for`-`in`-expressions remove the risk of off-by-one indexing errors, and thus, `for`-`in` **is to be preferred over** `for`-`to`.

Lists support slicing identically to strings, as demonstrated in **??**.

Lists may be concatenated using either the *"@" concatenation* operator or the *"::"*

Listing 12.17 listFor.fsx:
The for-in loops are preferred over for-to loops.

```
1  let printList (lst : int list) : unit =
2    for elm in lst do
3      printf "%A " elm
4    printfn ""
5
6  printList [3; 4; 5]
```
- -
```
1  $ fsharpc --nologo listFor.fsx && mono listFor.exe
2  3 4 5
```

---

cons operators. The difference is that "@" concatenates two lists of identical types, while "::" concatenates an element and a list of identical types. This is demonstrated in **??**. Since lists are represented as linked lists, the cons operator is very efficient and has computational complexity $\mathcal{O}(1)$, while concatenation has computational complexity $\mathcal{O}(n)$, where n is the length of the first list.

It is possible to make multidimensional lists as lists of lists, as shown in **??**. The example shows a *ragged multidimensional list*, since each row has a different number of elements. This is also illustrated in **??**.
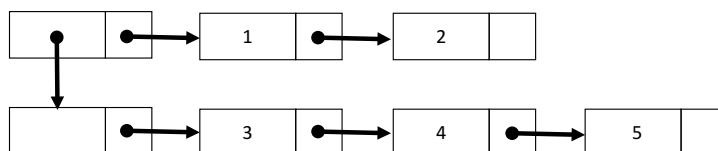
Figure 12.2: A list is a ragged linked list: Here is illustrated the linked list of
[[1;2];[3;4;5]].

The indexing of a particular element is slow due to the linked list implementation of lists, which is why arrays are often preferred for two- and higher-dimensional data structures, see **??**.

Listing 12.18: Examples of list slicing. Compare with ??.

```
1  > let lst = ['a' .. 'g'];;
2  val lst : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g']
3
4  > lst.[0];;
5  val it : char = 'a'
6
7  > lst.[3];;
8  val it : char = 'd'
9
10 > lst.[3..];;
11 val it : char list = ['d'; 'e'; 'f'; 'g']
12
13 > lst.[..3];;
14 val it : char list = ['a'; 'b'; 'c'; 'd']
15
16 > lst.[1..3];;
17 val it : char list = ['b'; 'c'; 'd']
18
19 > lst.[*];;
20 val it : char list = ['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g']
```

Listing 12.19: Examples of list concatenation.

```
1  > ([1] @ [2; 3]);;
2  val it : int list = [1; 2; 3]
3
4  > ([1; 2] @ [3; 4]);;
5  val it : int list = [1; 2; 3; 4]
6
7  > (1 :: [2; 3]);;
8  val it : int list = [1; 2; 3]
```

Listing 12.20 listMultidimensional.fsx:
A ragged multidimensional list, built as lists of lists, and its indexing.

```
1  let a = [[1;2];[3;4;5]]
2  let row = a.Item 0 in printfn "%A" row
3  let elm = row.Item 1 in printfn "%A" elm
4  let elm = (a.Item 0).Item 1 in printfn "%A" elm
```

```
1  $ fsharpc --nologo listMultidimensional.fsx && mono
     listMultidimensional.exe
2  [1; 2]
3  2
4  2
```

12.2.1 List Properties

Lists support a number of properties, some of which are listed below.

`Head:` Returns the first element of a list.

> **Listing 12.21:** Head

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

`IsEmpty:` Returns true if the list is empty.

> **Listing 12.22:** IsEmpty

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

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

> **Listing 12.23:** Length

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

`Tail:` Returns the list, except for its first element.

> **Listing 12.24:** Tail

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

12.2.2 The List Module

The built-in `List` module contains a wealth of functions for lists, some of which are briefly summarized below:

`List.collect: f:('T -> 'U list) -> lst:'T list -> 'U list.`
    Applies `f` to each element in `lst` and return a concatenated list of the results.

```
Listing 12.25: List.collect

1  > List.collect (fun elm -> [elm; elm; elm]) [1; 2; 3];;
2  val it : int list = [1; 1; 1; 2; 2; 2; 3; 3; 3]
```

`List.contains: elm:'T -> lst:'T list -> bool.`
    Returns true or false depending on whether or not `elm` is contained in `lst`.

```
Listing 12.26: List.contains


```

`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 12.27: List.filter

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

`List.find: f:('T -> bool) -> lst:'T list -> 'T.`
    Returns the first element of `lst` for which `f` is true.

```
Listing 12.28: List.find

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

`List.findIndex: f:('T -> bool) -> lst:'T list -> int.`

Returns the index of the first element of `lst` for which `f` is true.

> **Listing 12.29:** `List.findIndex`
>
> ```
> 1  > List.findIndex (fun x -> x = 'k') ['a' .. 'z'];;
> 2  val it : int = 10
> ```

`List.fold: f:('S -> 'T -> 'S) -> elm:'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+1`
   elements `List.fold` calculates:

$$f (\dots (f (f \ elm \ lst.[0]) \ lst.[1]) \dots) \ lst.[n].$$

> **Listing 12.30:** `List.fold`
>
> ```
> 1  > let addSquares acc elm = acc + elm*elm
> 2  - List.fold addSquares 0 [0 .. 9];;
> 3  val addSquares : acc:int -> elm:int -> int
> 4  val it : int = 285
> ```

`List.foldBack: f:('T -> 'S -> 'S) -> lst:'T list -> elm:'S -> 'S.`
   Updates an accumulator iteratively backwards by applying `f` to each element
   in `lst`. The initial value of the accumulator is `elm`. For exampel, when `lst`
   consists of `n+1` elements `List.foldBack` calculates:

$$f \ lst.[0] \ (f \ lst.[1] \ (\dots(f \ lst.[n] \ elm) \dots)).$$

> **Listing 12.31:** `List.foldBack`
>
> ```
> 1  > let addSquares elm acc = acc + elm*elm
> 2  - List.foldBack addSquares [0 .. 9] 0;;
> 3  val addSquares : elm:int -> acc:int -> int
> 4  val it : int = 285
> ```

`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 12.32: `List.forall`

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

`List.head: lst:'T list -> int.`
Returns the first element in `lst`. An exception is raised if `lst` is empty. See **??**
for more on exceptions.

Listing 12.33: `List.head`

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

`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 12.34: `List.init`

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

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

Listing 12.35: `List.isEmpty`

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

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

Listing 12.36: `List.iter`

```
```

`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 12.37:** `List.map`

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

`List.ofArray: arr:'T [] -> 'T list.`
Returns a list whose elements are the same as `arr`. See **??** for more on arrays.

> **Listing 12.38:** `List.ofArray`

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

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

> **Listing 12.39:** `List.rev`

```
1  > List.rev [1; 2; 3];;
2  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 12.40:** `List.sort`

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

`List.tail: 'T list -> 'T list.`
Returns a new list identical to `lst` but without its first element. An Exception is raised if `lst` is empty. See **??** for more on exceptions.

```
   Listing 12.41: List.tail

1  > List.tail [1; 2; 3];;
2  val it : int list = [2; 3]
3
4  > let a = [1; 2; 3] in List.tail a;;
5  val it : int list = [2; 3]
```

List.toArray: lst:'T list -> 'T [].
    Returns an array whose elements are the same as lst. See **??** for more on
    arrays.

```
   Listing 12.42: List.toArray

1  > List.toArray [1; 2; 3];;
2  val it : int [] = [|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 12.43: List.unzip

1  > List.unzip [(1, 'a'); (2, 'b'); (3, 'c')];;
2  val it : int list * char list = ([1; 2; 3], ['a'; 'b';
     'c'])
3
4  >
```

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 12.44: List.zip

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

12.3 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 12.45:   **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 12.46:   **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 **??**.   Notice that in spite of the missing

Listing 12.47 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
```

```
$ fsharpc --nologo arrayReassign.fsx && mono
    arrayReassign.exe
[|1; 2; 3; 4; 5|]
[|1; 4; 9; 16; 25|]
```

`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 $\mathcal{O}(1)$, 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 $\mathcal{O}(n)$, 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 **??**. As

Listing 12.48: Examples of array slicing. Compare with ?? and ??.

```
1  > let arr = [|'a' .. 'g'|];;
2  val arr : char [] = [|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'|]
3
4  > arr.[0];;
5  val it : char = 'a'
6
7  > arr.[3];;
8  val it : char = 'd'
9
10 > arr.[3..];;
11 val it : char [] = [|'d'; 'e'; 'f'; 'g'|]
12
13 > arr.[..3];;
14 val it : char [] = [|'a'; 'b'; 'c'; 'd'|]
15
16 > arr.[1..3];;
17 val it : char [] = [|'b'; 'c'; 'd'|]
18
19 > arr.[*];;
20 val it : char [] = [|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'|]
```

illustrated, the missing start or end index imply from the first or to the last element, respectively.

Arrays do not have explicit operator support for appending and concatenation, instead the `Array` namespace includes an `Array.append` function, as shown in **??**.

Arrays are *reference types*, meaning that identifiers are references and thus suffer from aliasing, as illustrated in **??**.

Listing 12.49 arrayAppend.fsx:
Two arrays are appended with `Array.append`.

```
1  let a = [|1; 2;|]
2  let b = [|3; 4; 5|]
3  let c = Array.append a b
4  printfn "%A, %A, %A" a b c
```

```
1  $ fsharpc --nologo arrayAppend.fsx && mono arrayAppend.exe
2  [|1; 2|], [|3; 4; 5|], [|1; 2; 3; 4; 5|]
```

Listing 12.50 arrayAliasing.fsx:
Arrays are reference types and suffer from aliasing.

```
1  let a = [|1; 2; 3|];
2  let b = a
3  a.[0] <- 0
4  printfn "a = %A, b = %A" a b;;
```

```
1  $ fsharpc --nologo arrayAliasing.fsx && mono
     arrayAliasing.exe
2  a = [|0; 2; 3|], b = [|0; 2; 3|]
```

12.3.1 Array Properties and Methods

Some important properties and methods for arrays are:

`Clone(): 'T [].`
    Returns a copy of the array.

Listing 12.51: Clone

```
1  > let a = [|1; 2; 3|];
2  - let b = a.Clone()
3  - a.[0] <- 0
4  - printfn "a = %A, b = %A" a b;;
5  a = [|0; 2; 3|], b = [|1; 2; 3|]
6  val a : int [] = [|0; 2; 3|]
7  val b : obj = [|1; 2; 3|]
8  val it : unit = ()
```

`Length: int.`
    Returns the number of elements in the array.

Listing 12.52: Length

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

12.3.2 The Array Module

There are quite a number of built-in procedures for arrays in the `Array` module, some of which are summarized below.

`Array.append: arr1:'T [] -> arr2:'T [] -> 'T [].`
    Creates an new array whose elements are a concatenated copy of `arr1` and `arr2`.

Listing 12.53: Array.append

```
1  > Array.append [|1; 2;|] [|3; 4; 5|];;
2  val it : int [] = [|1; 2; 3; 4; 5|]
```

`Array.contains: elm:'T -> arr:'T [] -> bool.`

Returns true if `arr` contains `elm`.

Listing 12.54: `Array.contains`

```
1  > Array.contains 3 [|1; 2; 3|];;
2  val it : bool = true
```

`Array.exists: f:('T -> bool) -> arr:'T [] -> bool`.
Returns true if any application of `f` evaluates to true when applied to the elements of `arr`.

Listing 12.55: `Array.exists`

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

`Array.filter: f:('T -> bool) -> arr:'T [] -> 'T []`.
Returns an array of elements from `arr` who evaluate to true when `f` is applied to them.

Listing 12.56: `Array.filter`

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

`Array.find: f:('T -> bool) -> arr:'T [] -> 'T`.
Returns the first element in `arr` for which `f` evaluates to true. The `KeyNotFoundException` exception is raised if no element is found. See **??** for more on exceptions.

Listing 12.57: `Array.find`

```
1  > Array.find (fun x -> x % 2 = 1) [|0 .. 9|];;
2  val it : int = 1
```

`Array.findIndex: f:('T -> bool) -> arr:'T [] -> int`.
Returns the index of the first element in in `arr` for which `f` evaluates to true. If none are found, then the `System.Collections.Generic.KeyNotFoundException` exception is raised. See **??** for more on exceptions.

> **Listing 12.58:** `Array.findIndex`

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

`Array.fold: f:('S -> 'T -> 'S) -> elm:'S -> arr:'T [] -> 'S.`
Updates an accumulator iteratively by applying `f` to each element in `arr`. The initial value of the accumulator is `elm`. For example, when `arr` consists of `n+1` elements `Array.fold` calculates:

`f (... (f (f elm arr.[0]) arr.[1]) ...) arr.[n]`.

> **Listing 12.59:** `Array.fold`

```
1  > let addSquares acc elm = acc + elm*elm
2  - Array.fold addSquares 0 [|0 .. 9|];;
3  val addSquares : acc:int -> elm:int -> int
4  val it : int = 285
```

`Array.foldBack: f:('T -> 'S -> 'S) -> arr:'T [] -> elm:'S -> 'S.`
Updates an accumulator iteratively backwards by applying `f` to each element in `arr`. The initial value of the accumulator is `elm`. For exampel, when `arr` consists of `n+1` elements `Array.foldBack` calculates:

`f arr.[0] (f arr.[1] (...(f arr.[n] elm) ...))`.

> **Listing 12.60:** `Array.foldBack`

```
1  > let addSquares elm acc = acc + elm*elm
2  - Array.foldBack addSquares [|0 .. 9|] 0;;
3  val addSquares : elm:int -> acc:int -> int
4  val it : int = 285
```

`Array.forall: f:('T -> bool) -> arr:'T [] -> bool.`
Returns true if `f` evaluates to true for every element in `arr`.

> **Listing 12.61:** `Array.forall`

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

`Array.init: m:int -> f:(int -> 'T) -> 'T [].`

Create an array with `m` elements and whose value is the result of applying `f` to the index of the element.

Listing 12.62: `Array.init`

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

`Array.isEmpty: arr:'T [] -> bool.`

Returns true if `arr` is empty.

Listing 12.63: `Array.isEmpty`

```
> Array.isEmpty [||];;
val it : bool = true
```

`Array.iter: f:('T -> unit) -> arr:'T [] -> unit.`

Applies `f` to each element of `arr`.

Listing 12.64: `Array.iter`

```
> Array.iter (fun x -> printfn "%A " x) [|0; 1; 2|];;
0
```

`Array.map: f:('T -> 'U) -> arr:'T [] -> 'U [].`

Creates an new array whose elements are the results of applying `f` to each of the elements of `arr`.

Listing 12.65: `Array.map`

`Array.ofList: lst:'T list -> 'T [].`

Creates an array whose elements are copied from `lst`.

> **Listing 12.66:** `Array.ofList`

```
1  > Array.ofList [1; 2; 3];;
2  val it : int [] = [|1; 2; 3|]
```

`Array.rev: arr:'T [] -> 'T [].`
    Creates a new array whose elements are identical to `arr` but in reverse order.

> **Listing 12.67:** `Array.rev`

```
1  > Array.rev [|1; 2; 3|];;
2  val it : int [] = [|3; 2; 1|]
```

`Array.sort: arr:'T[] -> 'T [].`
    Creates a new array with the same elements as in `arr` but in sorted order

> **Listing 12.68:** `Array.sort`

```
1  > Array.sort [|3; 1; 2|];;
2  val it : int [] = [|1; 2; 3|]
```

`Array.toList: arr:'T [] -> 'T list.`
    Creates a new list whose elements are copied from `arr`.

> **Listing 12.69:** `Array.toList`

```
1  > Array.toList [|1; 2; 3|];;
2  val it : int list = [1; 2; 3]
```

`Array.unzip: arr:('T1 * 'T2) [] -> 'T1 [] * 'T2 [].`
    Returns a pair of arrays of all the first elements and all the second elements of `arr`, respectively.

> **Listing 12.70:** `Array.unzip`

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

```
Array.zip: arr1:'T1 [] -> arr2:'T2 [] -> ('T1 * 'T2) [].
```
    Returns a list of pairs, where elements in `arr1` and `arr2` are iteratively paired.

> **Listing 12.71:** `Array.zip`
>
> ```
> 1  > Array.zip [|1; 2; 3|] [|'a'; 'b'; 'c'|];;
> 2  val it : (int * char) [] = [|(1, 'a'); (2, 'b'); (3,
>      'c')|]
> ```

12.4 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 **??** is a jagged array of increasing width.   Indexing

> **Listing 12.72 arrayJagged.fsx:**
> **An array of arrays. When row lengths are of non-equal elements, then it is a jagged array.**
>
> ```
> 1  let arr = [|[|1|]; [|1; 2|]; [|1; 2; 3|]|]
> 2
> 3  for row in arr do
> 4    for elm in row do
> 5      printf "%A " elm
> 6    printf "\n"
> ```
> -
> ```
> 1  $ fsharpc --nologo arrayJagged.fsx && mono arrayJagged.exe
> 2  1
> 3  1 2
> 4  1 2 3
> ```

arrays of arrays is done sequentially, in the sense that in the above example, 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 **??**.   Note that the `for-in` cannot be used in `pownArray`, e.g.,

    `for row in arr do for elm in row do elm <- pown elm p done done`,

since the iterator value `elm` is not mutable, even though `arr` is an array.

Listing 12.73 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  $ fsharpc --nologo arrayJaggedSquare.fsx && mono
     arrayJaggedSquare.exe
2    1   2   3   4
3    1   9  25  49
4    1  16  49 100
```

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 **??** to use `Array2D` gives a slightly simpler program, which is shown in **??**. 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 **??**. 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 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 **??**.

Listing 12.74 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
```

```
$ fsharpc --nologo array2D.fsx && mono array2D.exe
[[0; 3; 6; 9]
 [1; 4; 7; 10]
 [2; 5; 8; 11]]
```

Listing 12.75: Examples of Array2D slicing. Compare with ??.

```
> 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]]
```

12.4.1 The Array2D Module

There are quite a number of built-in procedures for arrays in the `Array2D` namespace, some of which are summarized below.

`copy: arr:'T [,] -> 'T [,].`
> Creates a new array whose elements are copied from `arr`.

Listing 12.76: `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 12.77: `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 12.78: `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 12.79: `Array2D.iter`

```
1  > let arr = Array2D.init 3 4 (fun i j -> i + 10 * j)
2  - Array2D.iter (fun elm -> printf "%A " elm) arr
3  - printfn "";;
4  0 10 20 30 1 11 21 31 2 12 22 32
5  val arr : int [,] = [[0; 10; 20; 30]
6                       [1; 11; 21; 31]
7                       [2; 12; 22; 32]]
8  val it : unit = ()
```

`length1: arr:'T [,] -> int`.
   Returns the length the first dimension of `arr`.

Listing 12.80: `Array2D.length1`

```
1  > let arr = Array2D.create 2 3 0.0 in Array2D.length1
     arr;;
2  val it : int = 2
```

`length2: arr:'T [,] -> int`.
   Returns the length of the second dimension of `arr`.

Listing 12.81: `Array2D.forall length2`

```
1  > let arr = Array2D.create 2 3 0.0 in Array2D.length2
     arr;;
2  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 12.82: `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 The Imperative Programming 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 affects *states*. In F#, states are mutable and immutable values, and they are affected by functions and procedures. An imperative program is typically identified as using:

Mutable values
> Mutable values are holders of states, they may change over time, and thus have a dynamic scope.

Procedures
> Procedures are functions that returns "()", 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 uses side-effects to communicate with the terminal.

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.

Mono state or stateless programs, as *functional programming*, can be seen as a subset of imperative programming and is discussed in **??**. *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 **??**.

An imperative program is like a Turing machine, a theoretical machine introduced by Alan Turing in 1936 [**?**]. 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 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 double 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 changes. 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.

13.1 Imperative Design

Programming is the act of solving a problem by writing a program to be executed on a computer. The imperative programming paradigm focuses on states. To solve a problem, you could work through the following list of actions:

1. Understand the problem. As Pólya described it, see Chapter 1, the first step in any solution is to understand the problem. A good trick to check whether you understand the problem, is to briefly describe it in your own words.

2. Identify the main values, variables, functions, and procedures needed. If the list of procedures is large, then you most likely should organize them in modules.

3. For each function and procedure, write a precise description of what it should do. This can conveniently be performed as an in-code comment for the procedure, using the F# XML documentation standard.

4. Make mockup functions and procedures using the intended types, but do not necessarily compute anything sensible. Run through examples in your mind, using this mockup program to identify any obvious oversights.

5. Write a suite of unit tests that tests the basic requirements for your code. The

unit tests should be runnable with your mockup code. Writing unit tests will also allow you to evaluate the usefulness of the code pieces as seen from an application point of view.

6. Replace the mockup functions in a prioritized order, i.e., write the must-have code before you write the nice-to-have code, while regularly running your unit tests to keep track of your progress.

7. Evaluate the code in relation to the desired goal, and reiterate earlier actions as needed until the task has been sufficiently completed.

8. Complete your documentation both in-code and outside to ensure that the intended user has sufficient knowledge to effectively use your program and to ensure that you or a fellow programmer will be able to maintain and extend the program in the future.

14 Recursion

Recursion is a central concept in F# and is used to control flow in loops without the `for` and `while` constructions. **??** illustrates the concept of an infinite loop with recursion.
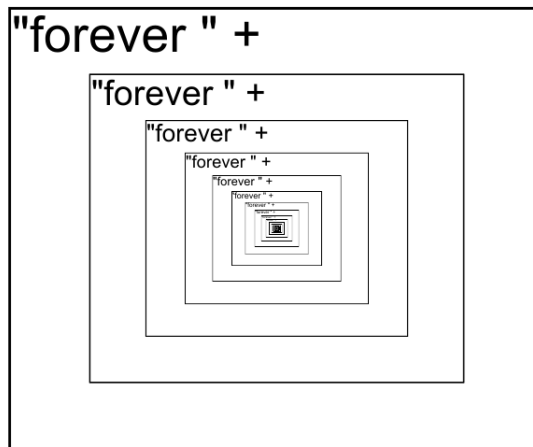
Figure 14.1: An infinitely long string of "forever forever forever...", conceptually calculated by `let rec` forever () = `"forever "` + (forever ()).

14.1 Recursive Functions

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

Listing 14.1: Syntax for defining one or more mutually dependent recursive functions.

```
let rec <ident> = <expr> {and <ident> = <expr>} [in] <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, then they must be defined jointly using the `and` keyword.

An example of a recursive function that counts from 1 to 10 similarly to **??** is given in **??**. Here the `prt` function calls itself repeatedly, such that the first call is `prt 1 10`,

Listing 14.2 countRecursive.fsx: Counting to 10 using recursion.

```
1  let rec prt a b =
2    if a > b then
3      printf "\n"
4    else
5      printf "%d " a
6      prt (a + 1) b
7
8  prt 1 10
```

```
1  $ fsharpc --nologo countRecursive.fsx && mono
     countRecursive.exe
2  1 2 3 4 5 6 7 8 9 10
```

which calls `prt 2 10`, and so on until the last call `prt 11 10`. Each time `prt` is called, new bindings named `a` and `b` are made to new values. This is illustrated in **??**. The old values are no longer accessible, as indicated by subscripts in the figure. E.g., in prt_3, the scope has access to a_3 but not a_2 and a_1. Thus, in this program, the process is similar to a `for` loop, where the counter is `a`, and in each loop its value is reduced.

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

Listing 14.3: Recursive functions consist of a stopping criterium, a stopping expression, and a recursive step.

```
1  let rec f a =
2    if <stopping condition>
3    then <stopping step>
4    else <recursion step>
```

The *match − with* are also very common conditional structures. In **??**, a > b is the *stopping condition*, `printfn "\n"` is the *stopping step*, and `printfn "\%d " a`;

```
$ fsharpc countRecursive.fsx && mono countRecursive.exe
```

```
prt 1 10
```

prt$_1$: a$_1$= 1, b$_1$= 10

```
1 > 10: ✗
  printf "1 "
  prt 2 10
```

prt$_2$: a$_2$= 2, b$_2$= 10

```
2 > 10: ✗
  printf "2 "
  prt 3 10
```

prt$_3$: a$_3$= 3, b$_3$= 10

```
3 > 10: ✗
  printf "3 "
  prt 4 10
    ⋰
      prt 11 10
```

prt$_{11}$: a$_{11}$= 1, b$_{11}$= 10

```
11 > 10 ✓
  printf "\\n"
```

```
()
```

```
    ⋰
```

```
()
```

```
  ()
```

```
()
```

```
()
```
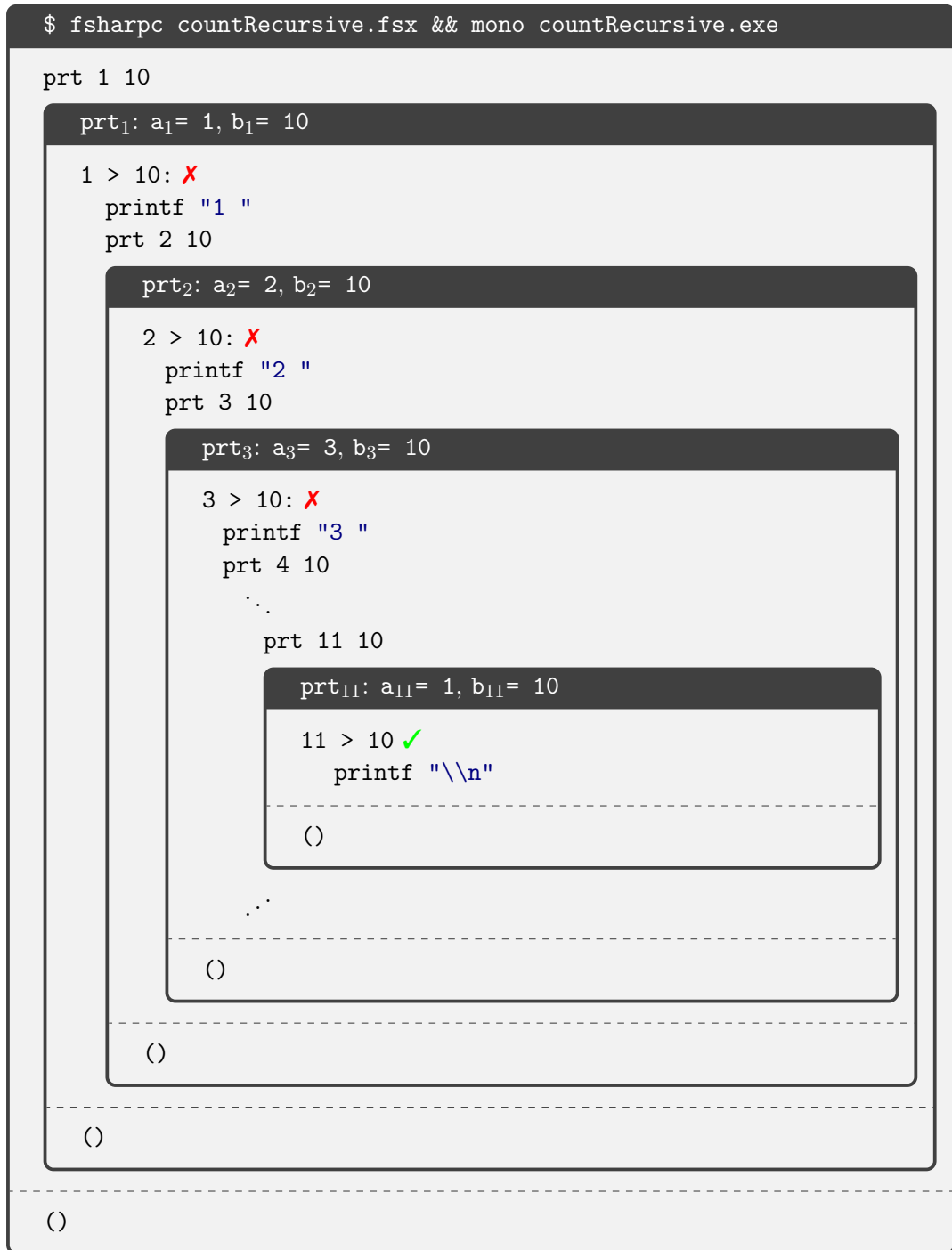
Figure 14.2: Illustration of the recursion used to write the sequence "1 2 3 ... 10" in line **??** in **??**. Each frame corresponds to a call to `prt`, where new values overshadow old ones. All calls return `unit`.

`prt (a + 1) b` is the *recursion step*.

14.2 The Call Stack and Tail Recursion

Fibonacci's sequence of numbers 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 **??**. Here we extended the sequence to $0, 1, 1, 2, 3, 5, \ldots$ with

Listing 14.4 fibRecursive.fsx:
The n'th Fibonacci number using recursion.

```
1  let rec fib n =
2    if n < 1 then
3        0
4    elif n = 1 then
5        1
6    else
7        fib (n - 1) + fib (n - 2)
8
9  for i = 0 to 10 do
10   printfn "fib(%d) = %d" i (fib i)
```

```
1  $ fsharpc --nologo fibRecursive.fsx && mono
     fibRecursive.exe
2  fib(0) = 0
3  fib(1) = 1
4  fib(2) = 1
5  fib(3) = 2
6  fib(4) = 3
7  fib(5) = 5
8  fib(6) = 8
9  fib(7) = 13
10 fib(8) = 21
11 fib(9) = 34
12 fib(10) = 55
```

the starting sequence $0, 1$, allowing us to define all $\text{fib}(n) = 0$, $n < 1$. Thus, our function is defined for all integers, and for the irrelevant negative arguments it fails gracefully by returning 0. This is a general piece of advice: **make functions that fail gracefully.**

A visualization of the calls and the scopes created by `fibRecursive` is shown in **??**. The figure illustrates that each recursive step results in two calls to the function, thus
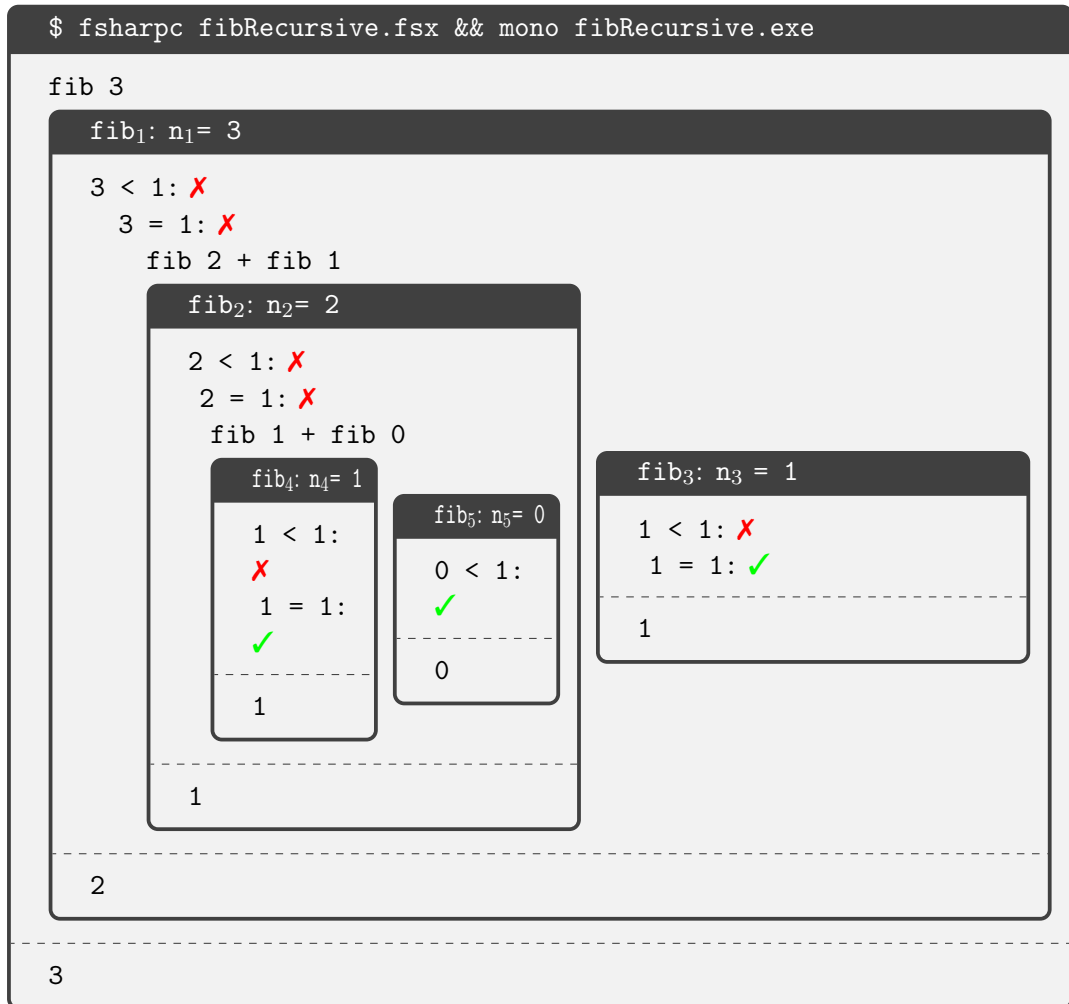
Figure 14.3: Illustration of the recursion used to write the sequence "1 2 3 ... 10" in line **??** in **??**. Each frame corresponds to a call to `fib`, where new values overshadow old ones.

creating two new scopes. And it gets worse. **??** 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 **??**, 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. This is a hugely ineffective implementation of calculating entries into Fibonacci's sequence, since many of the
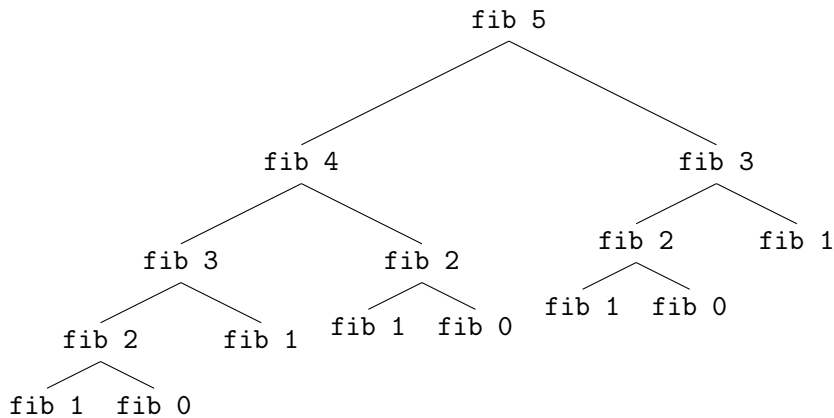
Figure 14.4: The function calls involved in calling `fib 5`.

calls are identical. E.g., in **??**, `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. **??** shows snapshots of the call stack when calling `fib 5` in **??**. The call first

Figure 14.5: A call to `fib 5` in **??** starts a sequence of function calls and stack frames on the call stack.

stacks a frame onto the call stack with everything needed to execute the 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 ?? $\mathcal{O}(\alpha^n)$, $\alpha = \frac{1+\sqrt{5}}{2}$ stacking operations are performed for a call to `fib n`. The $\mathcal{O}(f(n))$ is the *Landau symbol* used to denote the order of a function, such that if $g(n) = \mathcal{O}(f(n))$ 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 ??, 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 ?? can be improved to run faster and use less memory. One such algorithm is given in ?? Calculating the 45th Fibonacci

---

Listing 14.5 fibRecursiveAlt.fsx:
A fast, recursive implementation of Fibonacci's numbers. Compare with ??.

```
1   let fib n =
2     let rec fibPair n pair =
3       if n < 2 then pair
4       else fibPair (n - 1) (snd pair, fst pair + snd pair)
5     if n < 1 then 0
6     elif n = 1 then 1
7     else fibPair n (0, 1) |> snd
8
9   printfn "fib(10) = %d" (fib 10)
```

```
1   $ fsharpc --nologo fibRecursiveAlt.fsx && mono
      fibRecursiveAlt.exe
2   fib(10) = 55
```

---

number a MacBook Pro, with a 2.9 GHz Intel Core i5 using ?? takes about 11.2s while using ?? is about 224 times faster and only takes 0.050s. The reason is that `fib` in ?? calculates every number in 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.

?? also uses much less memory than ??, 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.**

14.3 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 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`. However, it is possible to implement mutual recursion by using functions as an argument, e.g., This being said, **??** is clearly to be preferred over **??**.

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: This is to be preferred anytime as the solution to the problem.

Listing 14.6 mutuallyRecursive.fsx:
Using mutual recursion to implement even and odd functions.

```
1   let rec even x =
2     if x = 0 then true
3     else odd (x - 1)
4   and odd x =
5     if x = 0 then false
6     else even (x - 1);;
7
8   let w = 5;
9   printfn "%*s %*s %*s" w "i" w "even" w "odd"
10  for i = 1 to w do
11    printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
1   $ fsharpc --nologo mutuallyRecursive.fsx && mono
      mutuallyRecursive.exe
2       i   even    odd
3       1 false    true
4       2   true false
5       3 false    true
6       4   true false
7       5 false    true
```

Listing 14.7 mutuallyRecursiveAlt.fsx:
Mutual recursion without the and **keyword requires a helper function.**

```
1  let rec evenHelper (notEven: int -> bool) x =
2    if x = 0 then true
3    else notEven (x - 1)
4
5  let rec odd x =
6    if x = 0 then false
7    else evenHelper odd (x - 1);;
8
9  let even x = evenHelper odd x
10
11 let w = 5;
12 printfn "%*s %*s %*s" w "i" w "Even" w "Odd"
13 for i = 1 to w do
14   printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
1  $ fsharpc --nologo mutuallyRecursiveAlt.fsx && mono
     mutuallyRecursiveAlt.exe
2      i  Even    Odd
3      1 false   true
4      2  true false
5      3 false   true
6      4  true false
7      5 false   true
```

Listing 14.8 parity.fsx:
A better way to test for parity without recursion.

```
1  let even x = (x % 2 = 0)
2  let odd x = not (even x)
```

14.4 Tracing Recursive Programs

Tracing by hand is a very illustrative method for understanding recursive programs. Consider the recursive program in **??**.   The program includes a function for cal-

Listing 14.9 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
```

culating the greatest common divisor of 2 integers, and calls this function with the numbers 10 and 15. Following the notation introduced in **??**, we write:

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

In line **??**, `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	??	E_0	gcd a b = ?
5	??	E_1	$\big((a = 10, b = 15), \text{gcd-body}, ()\big)$

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

Step	Line	Env.	Bindings and evaluations
6	??	E_1	gcd b a = ?
7	??	E_2	$\big((a = 15, b = 10), \text{gcd-body}, ()\big)$

In E_2, `a < b` in line **??** is false, but `b > 0` in line **??** 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
8	??	E_2	a % b = 5
9	??	E_2	gcd b (a % b) = ?
10	??	E_3	$\big((a = 10, b = 5), \text{gcd-body}, ()\big)$

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

Step	Line	Env.	Bindings and evaluations
11	??	E_3	a % b = 0
12	??	E_3	gcd b (a % b) = ?
13	??	E_4	$\big((a = 5, b = 0), \text{gcd-body}, ()\big)$

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

Step	Line	Env.	Bindings and evaluations
14	??	E_4	return = 5

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

Step	Line	Env.	Bindings and evaluations
15	??	E_3	return = 5

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

Step	Line	Env.	Bindings and evaluations
16	??	E_2	return = 5

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

Step	Line	Env.	Bindings and evaluations
17	??	E_1	return = 5

Finally, we scratch E_1, return to E_0, either physically or mentally replace the ?-mark with **5** and continue the evaluation of line **??**:

3 Step	Line	Env.	Bindings and evaluations
18	??	E_0	output = "gcd a b = 5"
19	??	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.

15 Programming with Types

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 **??**.

15.1 Type Abbreviations

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

```
  Listing 15.1:   Syntax for type abbreviation.

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

where the identifier is a new name, and the type-name is an existing type or a compound of existing types. **??** shows examples of the defintion 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 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.

Listing 15.2 typeAbbreviation.fsx:
Defining four type abbreviations, three of which are compound types.

```
1  type size = int
2  type position = float * float
3  type person = string * int
4  type intToFloat = int -> float
5
6  let sz : size = 3
7  let pos : position = (2.5, -3.2)
8  let pers : person = ("Jon", 50)
9  let conv : intToFloat = fun a -> float a
10 printfn "%A, %A, %A, %A" sz pos pers (conv 2)
```

```
1  $ fsharpc --nologo typeAbbreviation.fsx && mono
     typeAbbreviation.exe
2  3, (2.5, -3.2), ("Jon", 50), 2.0
```

15.2 Enumerations

Enumerations or *enums* for short are types with named values. Names in enums are assigned to a subset of integer or char values. Their syntax is as follows:

Listing 15.3:   Syntax for enumerations.

```
1  type <ident> =
2    [ | ] <ident> = <integerOrChar>
3    | <ident> = <integerOrChar>
4    | <ident> = <integerOrChar>
5    ...
```

An example of using enumerations is given in **??**.   In this example, we define an enumerated type for medals, which allows us to work with the names rather than the values. Since the values most often are arbitrary, we can program using semantically meaningful names instead. Being able to refer to an underlying integer type allows us to interface with other – typically low-level – programs that require integers, and to perform arithmetic. E.g., for the medal example, we can typecast the enumerated types to integers and calculate an average medal harvest.

Listing 15.4 enum.fsx:
An enum type acts as a typed alias to a set of integers or chars.

```
1  type medal =
2    Gold = 0
3    | Silver = 1
4    | Bronze = 2
5
6  let aMedal = medal.Gold
7  printfn "%A has value %d" aMedal (int aMedal)
```

```
1  $ fsharpc --nologo enum.fsx && mono enum.exe
2  Gold has value 0
```

15.3 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 15.5:   Syntax for type abbreviation.

```
1  [<attributes>]
2  type <ident> =
3    [| ]<ident> [of [<ident> :] <aType> [* [<ident> :] <aType>
       ...]]
4    | <ident> [of [<ident> :] <aType> [* [<ident> :] <aType>
       ...]]
5    ...
```

Discriminated unions are reference types, i.e., their content is stored on *The Heap*, see **??** for a discussion on reference types. 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. See **??** for a discussion on structs.

An example just using the named cases but no further specification of types is given in **??**. Here, we define a discriminated union as three named cases signifying three different types of medals. Comparing with the enumerated type in **??**, we see that the only difference is that the cases of the discriminated unions have no value. A

Listing 15.6 discriminatedUnions.fsx:
A discriminated union of medals. Compare with ??.

```
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
```

commonly used discriminated union is the *option type*, see **??** for more detail.

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 **??**. In this

---

Listing 15.7 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)
```

case, we define a discriminated union of two and three-dimensional vectors. 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 discrimated union values, as shown in Line **??**. When used, then the arguments may be given in arbitrary order, nevertheless, values for all fields must be given.

Discriminated unions can be defined recursively. This feature is demonstrated in **??**.

In this example we define a tree as depicted in **??**.

Listing 15.8 discriminatedUnionTree.fsx:
A discriminated union modelling binary trees.

```
1  type Tree =
2    Leaf of int
3    | Node of Tree * Tree
4
5  let one = Leaf 1
6  let two = Leaf 2
7  let three = Leaf 3
8  let tree = Node (Node (one, two), three)
9  printfn "%A" tree
```

```
1  $ fsharpc --nologo discriminatedUnionTree.fsx && mono
     discriminatedUnionTree.exe
2  Node (Node (Leaf 1, Leaf 2), Leaf 3)
```
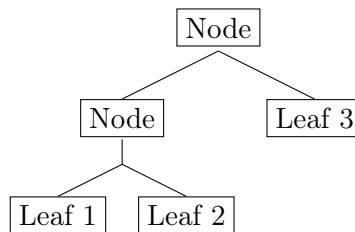
Figure 15.1: The tree with 3 leaves.

Pattern matching must be used in order to define functions on values of a discriminated union. E.g., in **??** we define a function that traverses a tree and prints the content of the nodes.

Discriminated unions are very powerful and can often be used instead of class hierarchies. Class hierarchies are discussed in **??**.

15.4 Records

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

> **Listing 15.9 discriminatedUnionPatternMatching.fsx:**
> **A discriminated union modelling binary trees.**
>
> ```
> type Tree = Leaf of int | Node of Tree * Tree
> let rec traverse (t : Tree) : string =
>     match t with
>       Leaf(v) -> string v
>       | Node(left, right) -> (traverse left) + ", " + (traverse
>    right)
>
> let tree = Node (Node (Leaf 1, Leaf 2), Leaf 3)
> printfn "%A: %s" tree (traverse tree)
> ```
> ```
> $ fsharpc --nologo discriminatedUnionPatternMatching.fsx
>    && mono discriminatedUnionPatternMatching.exe
> Node (Node (Leaf 1, Leaf 2), Leaf 3): 1, 2, 3
> ```

> **Listing 15.10:   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*, see **??** for a discussion on reference types. Records can also be *struct records* using the `[<Struct>]` attribute, in which case they are value types. See **??** for a discussion on structs. An example of using records is given in **??**. The values of individual members of a record are obtained using the *"."* notation This example illustrates a 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 **??**.   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 15.11 records.fsx:
A record is defined for holding information about a person.

```
1  type person = {
2      name : string
3      age : int
4      height : float
5  }
6
7  let author = {name = "Jon"; age = 50; height = 1.75}
8  printfn "%A\nname = %s" author author.name
```
- -
```
1  $ fsharpc --nologo records.fsx && mono records.exe
2  { name = "Jon"
3    age = 50
4    height = 1.75 }
5  name = Jon
```

`GetType()` method is inherited from the base class for all types, see **??** 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 to the individual members of the source or using the short-hand *with* notation. This is demonstrated in **??**. 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 **??**, 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 15.12 recordsDominance.fsx:
Redefined types dominate old record types, but earlier definitions are still accessible using 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
```

Listing 15.13 recordCopy.fsx:
Bindings are references. To copy and not make an alias, explicit copying must be performed.

```
1  type person = {
2    name : string;
3    mutable age : int;
4  }
5
6  let author = {name = "Jon"; age = 50}
7  let authorAlias = author
8  let authorCopy = {name = author.name; age = author.age}
9  let authorCopyAlt = {author with name = "Noj"}
10  author.age <- 51
11  printfn "author : %A" author
12  printfn "authorAlias : %A" authorAlias
13  printfn "authorCopy : %A" authorCopy
14  printfn "authorCopyAlt : %A" authorCopyAlt
```

```
1  $ fsharpc --nologo recordCopy.fsx && mono recordCopy.exe
2  author : { name = "Jon"
3    age = 51 }
4  authorAlias : { name = "Jon"
5    age = 51 }
6  authorCopy : { name = "Jon"
7    age = 50 }
8  authorCopyAlt : { name = "Noj"
9    age = 50 }
```

15.5 Structures

Structures, or *structs* for short, have much in common with records. They specify a
compound type with named fields, but they are value types, and they allow for some
customization of what is to happen when a value of its type is created. Since they are
value types, they are best used for small amounts of data. The syntax for defining
struct types are:

```
Listing 15.14:   Syntax for type abbreviation.

1  [ <attributes> ]
2  [<Struct>]
3  type <ident> =
4    val [ mutable ] <label1> : <type1>
5    val [ mutable ] <label2> : <type2>
6    ...
7    [new (<arg1>, <arg2>, ...) = {<label1> = <arg1>; <label1>
      = <arg2>; ...}
8    [new (<arg1>, <arg2>, ...) = {<label1> = <arg1>; <label1>
      = <arg2>; ...}
9    ...
```

The syntax makes use of the *val* and `new` keywords. Like `let`, the keyword `val`
binds a name to a value, but unlike `let`, the value is always the type's default value.
The `new` keyword denotes the function used to fill values into the fields at time of
creation. This function is called the *constructor*. No `let` or `do`-bindings are allowed
in structure definitions. Fields are accessed using the "*.*" notation. An example is
given in **??**.

```
Listing 15.15 struct.fsx:
Defining a struct type and creating a value of it.

1  [<Struct>]
2  type position =
3    val x : float
4    val y : float
5    new (a : float, b : float) = {x = a; y = b}
6
7  let p = position (3.0, 4.2)
8  printfn "%A: x = %A, y = %A" p p.x p.y
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
1  $ fsharpc --nologo struct.fsx && mono struct.exe
2  Struct+position: x = 3.0, y = 4.2
```

Structs are small versions of classes and allows, e.g., for overloading of the `new` constructor and for overriding of the inherited *ToString()* function. This is demonstrated in **??**. We defer further discussion of these concepts to **??**.

Listing 15.16 structOverloadNOverride.fsx:
Overloading the `new` **constructor and overriding the default** `ToString()`
function.

```
1  [<Struct >]
2  type position =
3    val x : float
4    val y : float
5    new (a : float , b : float) = {x = a; y = b}
6    new (a : int , b : int) = {x = float a; y = float b}
7    override this.ToString () =
8      "(" + (string this.x) + ", " + (string this.y) + ")"
9
10 let pFloat = position (3.0 , 4.2)
11 let pInt = position (3, 4)
12 printfn "%A and %A" pFloat pInt
```

```
1  $ fsharpc --nologo structOverloadNOverride.fsx && mono
     structOverloadNOverride.exe
2  (3, 4.2) and (3, 4)
```

The use of structs are generally discouraged, and instead, it is recommended to use enums, records, and discriminated unions, possibly with the `[<Struct>]` attribute for the last two in order to make them value types.

15.6 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 **??**. In this example, the 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.

Listing 15.17 variableType.fsx:
A function `apply` with runtime resolved types.

```
1  let apply (f : 'a -> 'a -> 'a) (x : 'a) (y : 'a) : 'a = f x y
2  let intPlus (x : int) (y : int) : int = x + y
3  let floatPlus (x : float) (y : float) : float = x + y
4
5  printfn "%A %A" (apply intPlus 1 2) (apply floatPlus 1.0 2.0)
```

```
1  $ fsharpc --nologo variableType.fsx && mono
     variableType.exe
2  3 3.0
```

The example in **??** 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 **??**? Unfortunately, the example fails to compile, since the type 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 **??**. In the example, adding 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 **??**. The example in **??** 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 identical type, and where the type can be inferred at compile time. See **??** 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 over. An alternative seems to be using runtime resolved variable types with the `'<ident>` syntax. Unfortunately, this is not possible in 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 (+))`.

Listing 15.18 variableTypeError.fsx:
Even though the "+" operator is defined for both integers and floats,
the type inference is static and infers plus : int -> int.

```
1  let plus x y = x + y
2
3  printfn "%A %A" (plus 1 2) (plus 1.0 2.0)
```

```
1  $ fsharpc --nologo variableTypeError.fsx && mono
     variableTypeError.exe
2
3  variableTypeError.fsx(3,34): error FS0001: This expression
     was expected to have type
4      'int'
5  but here has type
6      'float'
7
8  variableTypeError.fsx(3,38): error FS0001: This expression
     was expected to have type
9      'int'
10 but here has type
11     'float'
```

Listing 15.19 variableTypeInline.fsx:
The keyword inline forces static and independent inference each place
the function is used. Compare to the error case in ??.

```
1  let inline plus x y = x + y
2
3  printfn "%A %A" (plus 1 2) (plus 1.0 2.0)
```

```
1  $ fsharpc --nologo variableTypeInline.fsx && mono
     variableTypeInline.exe
2  3 3.0
```

Listing 15.20 compiletimeVariableType.fsx:
**Explicitly spelling out of the statically resolved type variables from
??.**

```
1  let inline plus (x : ^a) (y : ^a) : ^a when ^a : (static member (
      + ) :  ^a * ^a -> ^a) = x + y
2
3  printfn "%A %A" (plus 1 2) (plus 1.0 2.0)
```

```
1  $ fsharpc --nologo compiletimeVariableType.fsx && mono
      compiletimeVariableType.exe
2  3 3.0
```

16 Pattern Matching

Pattern matching is used to transform values and variables into a syntactical structure. The simplest example is value-bindings. The *let*-keyword was introduced in **??**, its extension with pattern matching is given as,

Listing 16.1: Syntax for let-expressions with pattern matching.

```
[[<Literal >]]
let [mutable] <pat> [: <returnType>] = <bodyExpr> [in
    <expr>]
```

A typical use of this is to extract elements of tuples, as demonstrated in **??**. Here

Listing 16.2 letPattern.fsx:
Patterns in let expressions may be used to extract elements of tuples.

```
let a = (3,4)
let (x,y) = a
let (alsoX,_) = a
printfn "%A: %d %d %d" a x y alsoX
```

```
$ fsharpc --nologo letPattern.fsx && mono letPattern.exe
(3, 4): 3 4 3
```

we extract the elements of a pair twice. First by binding to x and y, and second by binding to alsoX while using the wildcard pattern to ignore the second element. Thus, again the wildcard pattern in value-bindings is used to underline a disregarded value.

Another common use of patterns is as an alternative to if – then – else expressions, particularly when parsing input for a function. Consider the example in **??**. In the example, a discriminated union and a function are defined. The function converts each case to a supporting statement, using an if-expression. The same can be done with the *match – with* expression and patterns, as demonstrated in **??**. Here we used a pattern for the discriminated union cases and a wildcard pattern as default.

Listing 16.3 switch.fsx:
Using *if − then − else* to print discriminated unions.

```
1  type Medal = Gold | Silver | Bronze
2  let statement (m : Medal) : string =
3    if m = Gold then "You won"
4    elif m = Silver then "You almost won"
5    else "Maybe you will win next time"
6
7  let m = Silver
8  printfn "%A : %s" m (statement m)
```

```
1  $ fsharpc --nologo switch.fsx && mono switch.exe
2  Silver : You almost won
```

The lightweight syntax for `match`-expressions is,

Listing 16.5: Syntax for `match`-expressions.

```
1  match <inputExpr> with
2    [| ]<pat> [when <guardExpr>] -> <caseExpr>
3    | <pat> [when <guardExpr>] -> <caseExpr>
4    | <pat> [when <guardExpr>] -> <caseExpr>
5    ...
```

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. In lightweight syntax, 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.

Patterns are also used in a version of *for*-loop expressions, and its lightweight syntax is given as,

Listing 16.6: Syntax for `for`-expressions with pattern matching.

```
1  for <pat> in <sourceExpr> do
2    <bodyExpr>
```

Typically, `<sourceExpr>` is a list or an array. An example is given in **??**.   The

Listing 16.4 switchPattern.fsx:
Using match − with to print discriminated unions.

```
1  type Medal = Gold | Silver | Bronze
2  let statement (m : Medal) : string =
3    match m with
4      Gold -> "You won"
5      | Silver -> "You almost won"
6      | _ -> "Maybe you can win next time"
7
8  let m = Silver
9  printfn "%A : %s" m (statement m)
```

```
1  $ fsharpc --nologo switchPattern.fsx && mono
      switchPattern.exe
2  Silver : You almost won
```

Listing 16.7 forPattern.fsx:
Patterns may be used in for-loops.

```
1  for (_,y) in [(1,3); (2,1)] do
2    printfn "%d" y
```

```
1  $ fsharpc --nologo forPattern.fsx && mono forPattern.exe
2  3
3  1
```

wildcard pattern is used to disregard the first element in a pair while iterating over the complete list. It is good practice to **use wildcard patterns to emphasize unused values.**

The final expression involving patterns to be discussed is the *anonymous functions*. Patterns for anonymous functions have the syntax,

Listing 16.8: Syntax for anonymous functions with pattern matching.

```
1  fun <pat> [<pat> ...] -> <bodyExpr>
```

This is an extension of the syntax discussed in **??**. A typical use for patterns in *fun*-expressions is shown in **??**. Here we use an anonymous function expression and bind it to f. The expression has one argument of any type, which it ignores through the

Listing 16.9 funPattern.fsx:
Patterns may be used in fun**-expressions.**

```
1  let f = fun _ -> "hello"
2  printfn "%s" (f 3)
```
```
1  $ fsharpc --nologo funPattern.fsx && mono funPattern.exe
2  hello
```

wildcard pattern. Some limitations apply to the patterns allowed in fun-expressions. The wildcard pattern in fun-expressions are often used for *mockup functions*, where the code requires the said function, but its content has yet to be decided. Thus, mockup functions can be used as loose place-holders while experimenting with program design.

Patterns are also used in exceptions to be discussed in **??**, and in conjunction with the function-keyword, a keyword we discourage in this book. We will now demonstrate a list of important patterns in F#.

16.1 Wildcard Pattern

A *wildcard pattern* is denoted "_" and matches anything, see e.g., **??**.   In this

Listing 16.10 wildcardPattern.fsx:
Constant patterns match to constants.

```
1  let whatEver (x : int) : string =
2    match x with
3      _ -> "If you say so"
4
5  printfn "%s" (whatEver 42)
```
```
1  $ fsharpc --nologo wildcardPattern.fsx && mono
     wildcardPattern.exe
2  If you say so
```

example, anything matches the wildcard pattern, so all cases are covered and the function always returns the same sentence. This is rarely a useful structure on its own, since this could be replaced by a value binding or by a function ignoring its

input. However, wildcard patterns are extremely useful, since they act as the final `else` in `if`-expressions.

16.2 Constant and Literal Patterns

A *constant pattern* matches any input pattern with constants, see e.g., **??**. In this

---

Listing 16.11 constPattern.fsx:
Constant patterns match to constants.

```
1  type Medal = Gold | Silver | Bronze
2  let intToMedal (x : int) : Medal =
3    match x with
4      0 -> Gold
5      | 1 -> Silver
6      | _ -> Bronze
7
8  printfn "%A" (intToMedal 0)
```
- -
```
1  $ fsharpc --nologo constPattern.fsx && mono
     constPattern.exe
2  Gold
```

---

example, the input pattern is queried for a match with 0, 1, or the wildcard pattern. Any simple literal type constants may be used in the constant pattern, such as 8, 23y, 1010u, 1.2, `"hello world"`, `'c'`, and `false`. Here we also use the wildcard pattern. Note that matching is performed in a lazy manner and stops at the first matching case from the top. Thus, although the wildcard pattern matches everything, its case expression is only executed if none of the previous patterns match the input.

Constants can also be pre-bound by the `[<Literal>]` attribute for value-bindings. This is demonstrated in **??**. The attribute is used to identify the value-binding `TheAnswer` to be used, as if it were a simple literal type. Literal patterns must be either uppercase or module prefixed identifiers.

16.3 Variable Patterns

A *variable pattern* is a single lower-case letter identifier. Variable pattern identifiers are assigned the value and type of the input pattern. Combinations of constant and variable patterns are also allowed in conjunction with with records and arrays. This

---

Listing 16.12 literalPattern.fsx:
A variant of constant patterns is literal patterns.

```
1  [<Literal>]
2  let TheAnswer = 42
3  let whatIsTheQuestion (x : int) : string =
4    match x with
5      TheAnswer -> "We will need to build a bigger machine..."
6      | _ -> "Don't know that either"
7
8  printfn "%A" (whatIsTheQuestion 42)
```

```
1  $ fsharpc --nologo literalPattern.fsx && mono
     literalPattern.exe
2  "We will need to build a bigger machine..."
```

---

is demonstrated in **??**. In this example, the value identifier **n** has the function of a

---

Listing 16.13 variablePattern.fsx:
Variable patterns are useful for, e.g., extracting and naming fields

```
1  let (name, age) = ("Jon", 50)
2  let getAgeString (age : int) : string =
3    match age with
4      0 -> "a newborn"
5      | 1 -> "1 year old"
6      | n -> (string n) + " years old"
7
8  printfn "%s is %s" name (getAgeString age)
```

```
1  $ fsharpc --nologo variablePattern.fsx && mono
     variablePattern.exe
2  Jon is 50 years old
```

---

named wildcard pattern. Hence, the case could as well have been | _ -> (string age) + "years old", since age is already defined in this scope. However, variable patterns syntactically act as an argument to an anonymous function and thus act to isolate the dependencies. They are also very useful together with guards, see **??**.

16.4 Guards

A *guard* is a pattern used together with `match`-expressions including the *when*-keyword, as shown in **??**. Here guards are used to iteratively carve out subset of integers to as-

---

Listing 16.14 guardPattern.fsx:
Guard expressions can be used with other patterns to restrict matches.

```
1  let getAgeString (age : int) : string =
2    match age with
3      n when n < 1 -> "infant"
4      | n when n < 13 -> "child"
5      | n when n < 20 -> "teen"
6      | _ -> "adult"
7
8  printfn "A person aged %d is a/an %s" 50 (getAgeString 50)
```

```
1  $ fsharpc --nologo guardPattern.fsx && mono
     guardPattern.exe
2  A person aged 50 is a/an adult
```

---

sign different strings to each set. The guard expression in `<pat> when <guardExpr>` `-> <caseExpr>` is any expression evaluating to a Boolean, and the case expression is only executed for the matching case.

16.5 List Patterns

Lists have a concatenation pattern associated with them. The *"::"* cons-operator is used to to match the head and the rest of a list, and *"[]"* is used to match an empty list, which is also sometimes called the nil-case. This is very useful when recursively processing lists, as shown in **??** In the example, the function `sumList` uses the cons operator to match the head of the list with `n` and the tail with `rest`. The pattern `n :: tail` also matches `3 :: []`, and in that case `tail` would be assigned the value `[]`. When `lst` is empty, then it matches with "`[]`". List patterns can also be matched explicitly named elements, as demonstrated in the `sumThree` function. The elements to be matched can be any mix of constants and variables.

It is also possible to match on a series of cons-operators. For example `elm0 :: elm1 :: rest` would match a list with at least two elements, where the first will be bound to `elm1`, the second to `elm2`, and the remainder to `rest`.

Listing 16.15 listPattern.fsx:
Recursively parsing a list with list patterns.

```
1   let rec sumList (lst : int list) : int =
2     match lst with
3       n :: rest -> n + (sumList rest)
4       | [] -> 0
5
6   let rec sumThree (lst : int list) : int =
7     match lst with
8       [a; b; c] -> a + b + c
9       | _ -> sumList lst
10
11  let aList = [1; 2; 3]
12  printfn "The sum of %A is %d, %d" aList (sumList aList) (sumThree
      aList)
```

```
1   $ fsharpc --nologo listPattern.fsx && mono listPattern.exe
2   The sum of [1; 2; 3] is 6, 6
```

16.6 Array, Record, and Discriminated Union Patterns

Array, *record*, and *discriminated union patterns* are direct extensions on constant, variable, and wildcard patterns. **??** gives examples of array patterns. In the function `arrayToString`, the first case matches arrays of 3 elements where the first is the integer 1, the second case matches arrays of 3 elements where the second is a 1 and names the first `x`, and the final case matches all arrays and works as a default match case. As demonstrated, the cases are treated from first to last, and only the expression of the first case that matches is executed.

For record patterns, we use the field names to specify matching criteria. This is demonstrated in **??**. Here, the record type `Address` is created, and in the function `getZip`, a variable pattern `z` is created for naming zip values, and the remaining fields are ignored. Since the fields are named, the pattern match need not mention the ignored fields, and the example match is equivalent to `{zip = z} -> z`. The curly brackets are required for record patterns.

Discriminated union patterns are similar. For discriminated unions with arguments, the arguments can be matched as constants, variables, or wildcards. A demonstration is given in **??**. In the `project`-function, three-dimensional vectors are projected to two dimensions by removing the third element. Two-dimensional vectors are unchanged. The example uses the wildcard pattern to emphasize that the third element of three-dimensional vectors is ignored. Named arguments can also be matched, in

Listing 16.16 arrayPattern.fsx:
Using variable patterns to match on size and content of arrays.

```
1  let arrayToString (x : int []) : string =
2    match x with
3      [|1;_;_|] -> "3 elements, first of is 1"
4      | [|x;1;_|] -> "3 elements, first is " + (string x) + "
    Second 1"
5      | x -> "A general array"
6
7  printfn "%s" (arrayToString [|1; 1; 1|])
8  printfn "%s" (arrayToString [|3; 1; 1|])
9  printfn "%s" (arrayToString [|1|])
```

```
1  $ fsharpc --nologo arrayPattern.fsx && mono
     arrayPattern.exe
2  3 elements, first of is 1
3  3 elements, first is 3 Second 1
4  A general array
```

which case ";" is used instead of "," to delimit the fields in the match.

Listing 16.17 recordPattern.fsx:
Variable patterns for records to match on field values.

```
type Address = {street : string; zip : int; country : string}
let contact : Address = {
    street = "Universitetsparken 1";
    zip = 2100;
    country = "Denmark"}
let getZip (adr : Address) : int =
  match adr with
    {street = _; zip = z; country = _} -> z

printfn "The zip-code is: %d" (getZip contact)
```
```
$ fsharpc --nologo recordPattern.fsx && mono
    recordPattern.exe
The zip-code is: 2100
```

Listing 16.18 unionPattern.fsx:
Matching on discriminated union types.

```
type vector =
  Vec2D of float * float
  | Vec3D of float * float * float

let project (vec : vector) : vector =
  match vec with
    Vec3D (a, b, _) -> Vec2D (a, b)
    | v -> v

let v = Vec3D (1.0, -1.2, 0.9)
printfn "%A -> %A" v (project v)
```
```
$ fsharpc --nologo unionPattern.fsx && mono
    unionPattern.exe
Vec3D (1.0, -1.2, 0.9) -> Vec2D (1.0, -1.2)
```

16.7 Disjunctive and Conjunctive Patterns

Patterns may be combined using the "*/*" and "*&*" lexemes. These patterns are called disjunctive and conjunctive patterns, respectively, and work similarly to their logical operator counter parts, "`||`" and "`&&`".

Disjunctive patterns require at least one pattern to match, as illustrated in **??**. Here

---

Listing 16.19 disjunctivePattern.fsx:
Patterns can be combined logically as 'or' syntax structures.

```
1  let vowel (c : char) : bool =
2    match c with
3      'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
4      | _ -> false
5
6  String.iter (fun c -> printf "%A " (vowel c)) "abcdefg"
```

```
1  $ fsharpc --nologo disjunctivePattern.fsx && mono
     disjunctivePattern.exe
2  true false false false true false false
```

---

one or more cases must match for the final case expression, and thus, any vowel results in the value `true`. Everything else is matched with the wildcard pattern.

For *conjunctive patterns*, all patterns must match, which is illustrated in **??**.   In

---

Listing 16.20 conjunctivePattern.fsx:
Patterns can be combined logically as 'and' syntax structures.

```
1  let is11 (v : int * int) : bool =
2    match v with
3      (1,_) & (_,1) -> true
4      | _ -> false
5
6  printfn "%A" (List.map is11 [(0,0); (0,1); (1,0); (1,1)])
```

```
1  $ fsharpc --nologo conjunctivePattern.fsx && mono
     conjunctivePattern.exe
2  [false; false; false; true]
```

---

this case, we separately check the elements of a pair for the constant value `1` and return true only when both elements are `1`. In many cases, conjunctive patters can

be replaced by more elegant matches, e.g., using tuples, and in the above example a single case `(1,1) -> ` `true` would have been simpler. Nevertheless, conjunctive patterns are used together with active patterns, to be discussed below.

16.8 Active Patterns

The concept of patterns is extendable to functions. Such functions are called *active patterns*, and active patterns come in two flavors: regular and option types. The active pattern cases are constructed as function bindings, but using a special notation. They all take the pattern input as last argument, and may take further preceding arguments. The syntax for active patterns is one of,

```
Listing 16.21:   Syntax for binding active patterns to expressions.

1  let (|<caseName>|[_| ]) [ <arg> [<arg> ... ]]
     <inputArgument> = <expr>
2  let (|<caseName>|<caseName>|...|<caseName>|) <inputArgumet>
     = <expr>
```

When using the `(|<caseName>|_|])` variants, then the active pattern function must return an option type. `(|<caseName>|<caseName>|...|<caseName>|)` is the multi-case variant and must return a `Fsharp.Core.Choice` type. All other variants can return any type. There are no restrictions on arguments `<arg>`, and `<inputArgumetn>` is the input pattern to be matched. Notice in particular that the multi-case variant only takes one argument and cannot be combined with the option-type syntax. Below we will demonstrate by example how the various patterns are used.

The single case, `(|<caseName>|])`, matches all and is useful for extracting information from complex types, as demonstrated in **??**. Here we define a record to represent two-dimensional vectors and two different single case active patterns. Note that in the binding of the active pattern functions in line **??** and **??**, the argument is the input expression `match <inputExpr> with` ..., see **??**. However, the argument for the cases in line **??** and **??** are names bound to the output of the active pattern function.

Both `Cartesian` and `Polar` match a vector record, but they dismantle the contents differently. For an alternative solution using Class types, see **??**.

More complicated behavior is obtainable by supplying additional arguments to the single case. This is demonstrated in **??**. Here we supply an offset, which should be subtracted prior to calculating lengths and angles. Notice in line **??** that the

Listing 16.22 activePattern.fsx:
Single case active pattern for deconstructing complex types.

```fsharp
type vec = {x : float; y : float}
let (|Cartesian|) (v : vec) = (v.x, v.y)
let (|Polar|) (v : vec) = (sqrt(v.x*v.x + v.y * v.y), atan2 v.y
    v.x)
let printCartesian (p : vec) : unit =
    match p with
      Cartesian (x, y) -> printfn "%A:\n Cartesian (%A, %A)" p x y
let printPolar (p : vec) : unit =
    match p with
      Polar (a, d) -> printfn "%A:\n Polar (%A, %A)" p a d

let v = {x = 2.0; y = 3.0}
printCartesian v
printPolar v
```

```
$ fsharpc --nologo activePattern.fsx && mono
    activePattern.exe
{ x = 2.0
  y = 3.0 }:
 Cartesian (2.0, 3.0)
{ x = 2.0
  y = 3.0 }:
 Polar (3.605551275, 0.9827937232)
```

argument is given prior to the result binding.

Active pattern functions return option types are called *partial pattern functions*. The option type allows for specifying mismatches, as illustrated in **??**. In the example, we use the (|`<caseName>`|_|]) variant to indicate that the active pattern returns an option type. Nevertheless, the result binding `res` in line **??** uses the underlying value of `Some`. And in contrast to the two previous examples of single case patterns, the value `None` results in a mismatch. Thus in this case, if the denominator is `0.0`, then `Div res` does not match but the wildcard pattern does.

Multicase active patterns work similarly to discriminated unions without arguments. An example is given in **??**. In this example, we define three cases in line **??**. The result of the active pattern function must be one of these cases. For the `match`-expression, the match is based on the output of the active pattern function, hence in line **??**, the case expression is executed when the result of applying the active pattern function to the input expression `i` is `Gold`. In this case, a solution based on discriminated unions would probably be clearer.

Listing 16.23 activeArgumentsPattern.fsx:
All but the multi-case active pattern may take additional arguments.

```fsharp
type vec = {x : float; y : float}
let (|Polar|) (o : vec) (v : vec) =
  let x = v.x - o.x
  let y = v.y - o.y
  (sqrt(x*x + y * y), atan2 y x)
let printPolar (o : vec) (p : vec) : unit =
    match p with
      Polar o (a, d) -> printfn "%A:\n Cartesian (%A, %A)" p a d

let v = {x = 2.0; y = 3.0}
let offset = {x = 1.0; y = 1.0}
printPolar offset v
```

```
$ fsharpc --nologo activeArgumentsPattern.fsx && mono
   activeArgumentsPattern.exe
{ x = 2.0
  y = 3.0 }:
 Cartesian (2.236067977, 1.107148718)
```

Listing 16.24 activeOptionPattern.fsx:
Option type active patterns mismatch on `None` results.

```fsharp
let (|Div|_|) (e,d) = if d <> 0.0 then Some (e/d) else None

let safeDiv (p : float * float) =
    match p with
      | (0.0, 0.0) -> printfn "Div %A = undefined" p
      | Div res -> printfn "Div %A = %A" p res
      | _ -> printfn "Div %A = infinity" p

List.iter safeDiv [(1.0,1.0); (0.0,1.0); (1.0,0.0); (0.0,0.0)]
```

```
$ fsharpc --nologo activeOptionPattern.fsx && mono
   activeOptionPattern.exe
Div (1.0, 1.0) = 1.0
Div (0.0, 1.0) = 0.0
Div (1.0, 0.0) = infinity
Div (0.0, 0.0) = undefined
```

Listing 16.25 activeMultiCasePattern.fsx:
Multi-case active patterns have a syntactical structure similar to discriminated unions.

```fsharp
let (|Gold|Silver|Bronze|) inp =
  if inp = 0 then Gold
  elif inp = 1 then Silver
  else Bronze

let intToMedal (i : int) =
    match i with
      Gold -> printfn "%d: It's gold!" i
      | Silver -> printfn "%d: It's silver." i
      | Bronze -> printfn "%d: It's no more than bronze." i

List.iter intToMedal [0..3]
```

```
$ fsharpc --nologo activeMultiCasePattern.fsx && mono
   activeMultiCasePattern.exe
0: It's gold!
1: It's silver.
2: It's no more than bronze.
3: It's no more than bronze.
```

16.9 Static and Dynamic Type Pattern

Input patterns can also be matched on type. For *static type matching*, the matching is performed at compile time and indicated using the *":"* lexeme followed by the type name to be matched. Static type matching is further used as input to the type inference performed at compile time to infer non-specified types, as illustrated in **??**.

Here the head of the list `n` in the list pattern is explicitly matched as an integer,

---

Listing 16.26 staticTypePattern.fsx:
Static matching on type binds the type of other values by type inference.

```
1  let rec sum lst =
2      match lst with
3         (n : int) :: rest -> n + (sum rest)
4         | [] -> 0
5
6  printfn "The sum is %d" (sum [0..3])
```

```
1  $ fsharpc --nologo staticTypePattern.fsx && mono
     staticTypePattern.exe
2  The sum is 6
```

---

and the type inference system thus concludes that `lst` must be a list of integers.

In contrast to static type matching, *dynamic type matching* is performed at runtimes and indicated using the *":?"* lexeme followed by a type name. Dynamic type patterns allow for matching generic values at runtime. This is an advanced topic, which is included here for completeness. An example is given in **??**. In F#, all types are also objects whose type is denoted `obj`. Thus, the example uses the generic type when defining the argument to `isString`, and then dynamic type pattern matching for further processing. See **??** for more on objects. Dynamic type patterns are often used for analyzing exceptions, which is discussed in **??**. While dynamic type patterns are useful, they imply runtime checking, and **it is almost always better to prefer compile time over runtime type checking.**

Listing 16.27 dynamicTypePattern.fsx:
Dynamic matching on type binds the type of other values by type inference.

```fsharp
let isString (x : obj) : bool =
    match x with
        :? string -> true
        | _ -> false

let a = "hej"
printfn "Is %A a string? %b" a (isString a)
let b = 3
printfn "Is %A a string? %b" b (isString b)
```

```
$ fsharpc --nologo dynamicTypePattern.fsx && mono
    dynamicTypePattern.exe
Is "hej" a string? true
Is 3 a string? false
```

17 Higher-Order Functions

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. F# is a functions-first programming language with strong support for working with functions as values: Functions evaluate as *closures*, see **??**, 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 **??**. Here `List.map` applies the function `inc` to every element of the list.

Listing 17.1 higherOrderMap.fsx:
`List.map` **is a higher-order function, since it takes a function as argument.**

```
1  let inc x = x + 1
2  let newList = List.map inc [2; 3; 5]
3  printfn "%A" newList
```

```
1  $ fsharpc --nologo higherOrderMap.fsx && mono
     higherOrderMap.exe
2  [3; 4; 6]
```

---

higher-order functions are often used together with *anonymous functions*, where the anonymous functions is given as argument. For example, **??** may be rewritten using an anonymous function as shown in **??**. The code may be compacted even further, as shown in **??**. What was originally three lines in **??** including bindings to the names `inc` and `newList` has in **??** been reduced to a single line with no bindings. 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 allows 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. Further, for compact programs like **??**, it is not possible to perform a unit test of the function arguments. Finally, bindings emphasize semantic aspects of the evaluation 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

> **Listing 17.2 higherOrderAnonymous.fsx:**
> **An anonymous function is a higher-order function used here as an unnamed argument. Compare with ??.**
>
> ```
> 1  let newList = List.map (fun x -> x + 1) [2; 3; 5]
> 2  printfn "%A" newList
> ```
> --
> ```
> 1  $ fsharpc --nologo higherOrderAnonymous.fsx && mono
>       higherOrderAnonymous.exe
> 2  [3; 4; 6]
> ```

> **Listing 17.3 higherOrderAnonymousBrief.fsx:**
> **A compact version of ??.**
>
> ```
> 1  printfn "%A" (List.map (fun x -> x + 1) [2; 3; 5])
> ```
> --
> ```
> 1  $ fsharpc --nologo higherOrderAnonymousBrief.fsx && mono
>       higherOrderAnonymousBrief.exe
> 2  [3; 4; 6]
> ```

meaningful, but many programmers will find that the short is to be preferred in order to reduce the amount of typing to be performed.

Anonymous functions are also useful as return values of functions, as shown in **??**. Here the `inc` function produces a customized incrementation function as argument to `List.map`: It adds a prespecified number to an integer argument. Note that the closure of this customized function is only produced once, when the arguments for `List.map` is prepared, and not every time `List.map` maps the function to the elements of the list. Compare with **??**.

Piping is another example of a set of higher-order function: (<|), (|>), (<||), (||>), (<|||), (|||>). E.g., the functional equivalent of the right-to-left piping operator takes a value and a function and applies the function to the value, as demonstrated in **??**. Here the piping operator is used to apply the `inc` function to `aValue`. A more elegant way to write this would be `aValue |> inc`, or even just `inc aValue`.

> **Listing 17.4 higherOrderReturn.fsx:**
> **The procedure `inc` returns an increment function. Compare with ??.**
>
> ```
> 1  let inc n =
> 2    fun x -> x + n
> 3  printfn "%A" (List.map (inc 1) [2; 3; 5])
> ```
> --
> ```
> 1  $ fsharpc --nologo higherOrderReturn.fsx && mono
>      higherOrderReturn.exe
> 2  [3; 4; 6]
> ```

> **Listing 17.5 higherOrderPiping.fsx:**
> **The functional equivalent of the right-to-left piping operator is a higher-order function.**
>
> ```
> 1  let inc x = x + 1
> 2  let aValue = 2
> 3  let anotherValue = (|>) aValue inc
> 4  printfn "%d -> %d" aValue anotherValue
> ```
> --
> ```
> 1  $ fsharpc --nologo higherOrderPiping.fsx && mono
>      higherOrderPiping.exe
> 2  2 -> 3
> ```

17.1 Function Composition

Piping is a useful shorthand for composing functions, where the focus is on the transformation of arguments and results. Using higher-order functions, we can forgo the arguments and compose functions as functions directly. This is done with the ">>" and "<<" operators. An example is given in **??**. In the example we see that (`f >> g`) `x` gives the same result as `g (f x)`, while (`f << g`) `x` gives the same result as `f (g x)`. A memo technique for remembering the order of the application, when using the function composition operators, is that (`f >> g`) `x` is the same as `x |> f |> g`, i.e., the result of applying `f` to `x` is the argument to `g`. However, there is a clear distinction between the piping and composition operators. The type of the piping operator is

```
(|>) : ('a, 'a -> 'b) -> 'b
```

i.e., the piping operator takes a value of type `'a` and a function of type `'a -> 'b`, applies the function to the value, and produces the value `'b`. In contrast, the

> **Listing 17.6 higherOrderComposition.fsx:**
> **Functions defined as compositions of other functions.**

```
1  let f x = x + 1
2  let g x = x * x
3  let h = f >> g
4  let k = f << g
5  printfn "%d" (g (f 2))
6  printfn "%d" (h 2)
7  printfn "%d" (f (g 2))
8  printfn "%d" (k 2)
```

```
1  $ fsharpc --nologo higherOrderComposition.fsx && mono
     higherOrderComposition.exe
2  9
3  9
4  5
5  5
```

composition operator has type

```
(>>) : ('a -> 'b, 'b -> 'c) -> ('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`.

17.2 Currying

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 called *partial specification* or *currying* in tribute of Haskell Curry[1].
An example is given in **??**. Here, `mul 2.0` is a partial application of the function
`mul x y`, where the first argument is fixed, and hence `timesTwo` is a function of 1
argument being the second argument of `mul`. The same can be achieved using tuple
arguments, as shown in **??**. Conversion between multiple and tuple arguments is
easily done with higher-order functions, as demonstrated in **??**. Conversion between
multiple and tuple arguments are useful when working with higher-order functions
such as `Lst.map`. E.g., if `let mul (x, y) = x * y` as in **??**, then `curry mul` has the

---

[1]Haskell Curry (1900–1982) was an American mathematician and logician who also has a pro-
gramming language named after him: Haskell.

> **Listing 17.7 higherOrderCurrying.fsx:**
> **Currying: defining a function as a partial specification of another.**
>
> ```
> 1  let mul x y = x*y
> 2  let timesTwo = mul 2.0
> 3  printfn "%g" (mul 5.0 3.0)
> 4  printfn "%g" (timesTwo 3.0)
> ```
> --
> ```
> 1  $ fsharpc --nologo higherOrderCurrying.fsx && mono
>       higherOrderCurrying.exe
> 2  15
> 3  6
> ```

> **Listing 17.8 higherOrderTuples.fsx:**
> **Partial specification of functions using tuples is less elegant. Compare**
> **with ??.**
>
> ```
> 1  let mul (x, y) = x*y
> 2  let timesTwo y = mul (2.0, y)
> 3  printfn "%g" (mul (5.0, 3.0))
> 4  printfn "%g" (timesTwo 3.0)
> ```
> --
> ```
> 1  $ fsharpc --nologo higherOrderTuples.fsx && mono
>       higherOrderTuples.exe
> 2  15
> 3  6
> ```

type `x:'a -> y:'b -> 'c` as can be seen in **??**, and thus is equal to the anonymous
function `fun x y -> x * y`. Hence, `curry mul 2.0` is equal to `fun y -> 2.0 * y`,
since the precedence of function calls is `(curry mul) 2.0`.

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.**

Listing 17.9: Two functions to convert between two and 2-tuple arguments.

```
> let curry f x y = f (x,y)
- let uncurry f (x,y) = f x y;;
val curry : f:('a * 'b -> 'c) -> x:'a -> y:'b -> 'c
val uncurry : f:('a -> 'b -> 'c) -> x:'a * y:'b -> 'c
```

18 The Functional Programming Paradigm

Functional programming is a style of programming which performs computations by evaluating functions. Functional programming avoids mutable values and side-effects. It is declarative in nature, e.g., by the use of value- and function-bindings – `let`-bindings – and avoids statements – `do`-bindings. Thus, the result of a function in functional programming depends only on its arguments, and therefore functions have no side-effect and are deterministic, such that 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 **??**. 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 [**?**]. 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.

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 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.

18.1 Functional Design

A key to all good programming designs is encapsulating code into modules. For functional programs, the essence is to consider data and functions as transformations of data. I.e., the basic pattern is a piping sequence,

```
x |> f |> g |> h,
```

where `x` is the input data and `f`, `g`, and `h` are functions that transform the data. Of course, most long programs include lots of control structure, implying that we would need junctions in the pipe system, however, piping is a useful memo technique.

In functional programming there are some pitfalls that you should avoid:

- Creating large data structures, such as a single record containing all data. Since data is immutable, changing a single field in a monstrous record would mean a lot of copying in many parts of your program. In such cases, it is better to use a range of data structures that express isolated semantic units of your problem.

- Non-tail recursion. Relying on the built-in functions `map`, `fold`, etc., is a good start for efficiency.

- Single character identifiers. Since functional programming tends to produce small, well-defined functions, there is a tendency to use single character identifiers, e.g., `let f x = ....` In the very small, this can be defended, but the names used as identifiers can be used to increase the readability of code to yourself or to others. Typically, identifiers are long and informative in the outermost scope, while decreasing in size as you move in.

- Few comments. Since functional programming is very concise, there is a tendency for us as programmers to forget to add sufficient comments to the code, since at the time of writing, the meaning may be very clear and well thought through. However, experience shows that this clarity deteriorates fast with time.

- Identifiers that are meaningless clones of each other. Since identifiers cannot be reused except by overshadowing in deeper scopes, there is often a tendency to have a family of identifiers like `a`, `a2`, `newA` etc. It is better to use names

that more clearly state the semantic meaning of the values, or, if only used as temporary storage, to discard them completely in lieu of piping and function composition. However, the lattermost often requires comments describing the transformation being performed.

Thus, a design pattern for functional programs must focus on,

- What input data is to be processed

- How the data is to be transformed

For large programs, the design principle is often similar to other paradigms, which are often visualized graphically as components that take input, interact, and produce results often together with a user. The effect of functional programming is mostly seen in the small, i.e., where a subtask is to be structured functionally.

19 Handling Errors and Exceptions

19.1 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 **??**. The error message contains much information.

```
1  > 3 / 0;;
2  System.DivideByZeroException: Attempted to divide by zero.
3    at <StartupCode$FSI_0002>.$FSI_0002.main@ () [0x00000]
     in <32e12432747247dcb696ff61fd980cd3>:0
4    at (wrapper managed-to-native)
     System.Reflection.RuntimeMethodInfo.InternalInvoke(System.Reflection
5    at System.Reflection.RuntimeMethodInfo.Invoke
     (System.Object obj, System.Reflection.BindingFlags
     invokeAttr, System.Reflection.Binder binder,
     System.Object[] parameters,
     System.Globalization.CultureInfo culture) [0x0006a] in
     <e068e2227ab74c1bb3d724ebaab0e3ff>:0
6  Stopped due to error
```

Listing 19.1: Division by zero halts execution with an error message.

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 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 **??**.

Exceptions are handled by the `try`–keyword expressions. We say that an expression

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 19.1: Some built-in exceptions. The prefix `System.` has been omitted for brevity.

may *raise* or *cast* an exception and that the try–expression may *catch* and *handle* the exception by another expression.

Exceptions like in **??** may be handled by try–with expressions, as demonstrated in **??**. In the example, when the division operator raises the `System.DivideByZeroException`

Listing 19.2 exceptionDivByZero.fsx:
A division by zero is caught and a default value is returned.

```
1  let div enum denom =
2    try
3      enum / denom
4    with
5      | :? System.DivideByZeroException -> System.Int32.MaxValue
6
7  printfn "3 / 1 = %d" (div 3 1)
8  printfn "3 / 0 = %d" (div 3 0)
```

```
1  $ fsharpc --nologo exceptionDivByZero.fsx && mono
     exceptionDivByZero.exe
2  3 / 1 = 3
3  3 / 0 = 2147483647
```

---

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 19.3: Syntax for the try–with exception handling.

```
1  try
2      <testExpr>
3  with
4      [ | ] <pat1> -> <exprHndl1>
5      | <pa2> -> <exprHndl2>
6      | <pat3> -> <exprHndl3>
7      ...
```

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 **??** *dynamic type matching* is used (see **??**) 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 **??**. Here the exception value is bound to the name

Listing 19.4 exceptionDivByZeroNamed.fsx:
Exception value is bound to a name. Compare to ??.

```
1  let div enum denom =
2    try
3      enum / denom
4    with
5      | :? System.DivideByZeroException as ex ->
6        printfn "Error: %s" ex.Message
7        System.Int32.MaxValue
8
9  printfn "3 / 1 = %d" (div 3 1)
10 printfn "3 / 0 = %d" (div 3 0)
```
```
1  $ fsharpc --nologo exceptionDivByZeroNamed.fsx && mono
     exceptionDivByZeroNamed.exe
2  3 / 1 = 3
3  Error: Attempted to divide by zero.
4  3 / 0 = 2147483647
```

`ex`.

All exceptions may be caught as the dynamic type `System.Exception`, and F# im-

plements a short-hand for catching an exception and binding its value to a name as demonstrated in **??** Finally, the short-hand may be guarded with a *when*–guard,

---

Listing 19.5 exceptionDivByZeroShortHand.fsx:
An exception of type System.Exception is bound to a name. Compare to ??.

```
1  let div enum denom =
2    try
3      enum / denom
4    with
5      | ex -> printfn "Error: %s" ex.Message; System.Int32.MaxValue
6
7  printfn "3 / 1 = %d" (div 3 1)
8  printfn "3 / 0 = %d" (div 3 0)
```

```
1  $ fsharpc --nologo exceptionDivByZeroShortHand.fsx && mono
     exceptionDivByZeroShortHand.exe
2  3 / 1 = 3
3  Error: Attempted to divide by zero.
4  3 / 0 = 2147483647
```

---

as demonstrated in **??**. The first pattern only matches the System.Exception exception when enum is 0, in which case the exception handler returns 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 **??** and **??**, but if you would like to distinguish between types of exceptions, then you must use explicit type matching and possibly value binding demonstrated in **??** and **??**

The *try–finally* expression has the following syntax,

Listing 19.7: Syntax for the try–finally exception handling.

```
1  try
2      <testExpr>
3  finally
4      <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 **??**. 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

Listing 19.6 exceptionDivByZeroGuard.fsx:
An exception of type `System.Exception` **is bound to a name and guarded. Compare to ??.**

```
1  let div enum denom =
2    try
3      enum / denom
4    with
5      | ex when enum = 0 -> 0
6      | ex -> System.Int32.MaxValue
7
8  printfn "3 / 1 = %d" (div 3 1)
9  printfn "3 / 0 = %d" (div 3 0)
10 printfn "0 / 0 = %d" (div 0 0)
```

```
1  $ fsharpc --nologo exceptionDivByZeroGuard.fsx && mono
     exceptionDivByZeroGuard.exe
2  3 / 1 = 3
3  3 / 0 = 2147483647
4  0 / 0 = 0
```

is no outer `try`–`with` expression, then execution stops without having evaluated the `finally` branch.

Exceptions can be raised using the ***raise***-function,

Listing 19.9:   Syntax for the `raise` **function that raises exceptions.**

```
1  raise (<expr>)
```

An example of raising the `System.ArgumentException` is shown in **??** In this example, division by zero is never attempted and instead an exception is raised 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 19.8 exceptionDivByZeroFinally.fsx:
The `finally` **branch is executed regardless of an exception.**

```
1  let div enum denom =
2     printf "Doing division:"
3     try
4        printf " %d %d." enum denom
5        enum / denom
6     finally
7        printfn " Division finished."
8
9  printfn "3 / 1 = %d" (try div 3 1 with ex -> 0)
10 printfn "3 / 0 = %d" (try div 3 0 with ex -> 0)
```

```
1  $ fsharpc --nologo exceptionDivByZeroFinally.fsx && mono
      exceptionDivByZeroFinally.exe
2  Doing division: 3 1. Division finished.
3  3 / 1 = 3
4  Doing division: 3 0. Division finished.
5  3 / 0 = 0
```

Listing 19.11:   Syntax for defining new exceptions.

```
1  exception <ident> of <typeId> {* <typeId>}
```

An example of defining a new exception and raising it is given in **??**.   Here 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 **??**,
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 **??**.

F# includes the *failwith* function to simplify the most common use of excep-
tions.   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 **??**.
 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 **??**.   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 **??**, or use the built-in `Failure` pattern as
in **??**.   Notice how only the `Failure` pattern allows for the parsing of the message
given to `failwith` as an argument.

Listing 19.10 raiseArgumentException.fsx:
Raising the division by zero with customized message.

```
1  let div enum denom =
2    if denom = 0 then
3      raise (System.ArgumentException "Error: \"division by 0\"")
4    else
5      enum / denom
6
7  printfn "3 / 0 = %s" (try (div 3 0 |> string) with ex ->
     ex.Message)
```

```
1  $ fsharpc --nologo raiseArgumentException.fsx && mono
     raiseArgumentException.exe
2  3 / 0 = Error: "division by 0"
```

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 **??**.   The `invalidArg` function raises an `System.ArgumentException`, as shown in **??**.

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 **??**.   The `reraise` function is only allowed to be the final call in the expression of a `with` rule.

Listing 19.12 exceptionDefinition.fsx:
A user-defined exception is raised but not caught by outer construct.

```
1  exception DontLikeFive of string
2
3  let picky a =
4    if a = 5 then
5      raise (DontLikeFive "5 sucks")
6    else
7      a
8
9  printfn "picky %A = %A" 3 (try picky 3 |> string with ex ->
     ex.Message)
10 printfn "picky %A = %A" 5 (try picky 5 |> string with ex ->
     ex.Message)
```

```
1  $ fsharpc --nologo exceptionDefinition.fsx && mono
     exceptionDefinition.exe
2  picky 3 = "3"
3  picky 5 = "Exception of type
     'ExceptionDefinition+DontLikeFive' was thrown."
```

Listing 19.13 exceptionDefinitionNCatch.fsx:
Catching a user-defined exception.

```
1  exception DontLikeFive of string
2
3  let picky a =
4    if a = 5 then
5      raise (DontLikeFive "5 sucks")
6    else
7      a
8
9  try
10   printfn "picky %A = %A" 3 (picky 3)
11   printfn "picky %A = %A" 5 (picky 5)
12 with
13   | DontLikeFive msg -> printfn "Exception caught with message:
     %s" msg
```

```
1  $ fsharpc --nologo exceptionDefinitionNCatch.fsx && mono
     exceptionDefinitionNCatch.exe
2  picky 3 = 3
3  Exception caught with message: 5 sucks
```

Listing 19.14 exceptionFailwith.fsx:
An exception raised by `failwith`.

```
1  if true then failwith "hej"
```

```
1  $ fsharpc --nologo exceptionFailwith.fsx && mono
      exceptionFailwith.exe
2
3  Unhandled Exception:
4  System.Exception: hej
5     at
      <StartupCode$exceptionFailwith>.$ExceptionFailwith$fsx.main@
      () [0x0000b] in <62028a4fddd05a03a74503834f8a0262>:0
6  [ERROR] FATAL UNHANDLED EXCEPTION: System.Exception: hej
7     at
      <StartupCode$exceptionFailwith>.$ExceptionFailwith$fsx.main@
      () [0x0000b] in <62028a4fddd05a03a74503834f8a0262>:0
```

Listing 19.15 exceptionSystemException.fsx:
Catching a `failwith` exception using type matching pattern.

```
1  let _ =
2    try
3      failwith "Arrrrg"
4    with
5      :? System.Exception -> printfn "So failed"
```

```
1  $ fsharpc --nologo exceptionSystemException.fsx && mono
      exceptionSystemException.exe
2
3  exceptionSystemException.fsx(5,5): warning FS0067: This
      type test or downcast will always hold
4
5  exceptionSystemException.fsx(5,5): warning FS0067: This
      type test or downcast will always hold
6  So failed
```

Listing 19.16 exceptionMatchWildcard.fsx:
Catching a `failwith` exception using the wildcard pattern.

```
1  let _ =
2    try
3      failwith "Arrrrg"
4    with
5      _ -> printfn "So failed"
```

```
1  $ fsharpc --nologo exceptionMatchWildcard.fsx && mono
     exceptionMatchWildcard.exe
2  So failed
```

Listing 19.17 exceptionFailure.fsx:
Catching a `failwith` exception using the `Failure` pattern.

```
1  let _ =
2    try
3      failwith "Arrrrg"
4    with
5      Failure msg ->
6        printfn "The castle of %A" msg
```

```
1  $ fsharpc --nologo exceptionFailure.fsx && mono
     exceptionFailure.exe
2  The castle of "Arrrrg"
```

Listing 19.18 exceptionInvalidArg.fsx:
An exception raised by `invalidArg`. Compare with ??.

```
1  if true then invalidArg "a" "is too much 'a'"
```

```
1  $ fsharpc --nologo exceptionInvalidArg.fsx && mono
      exceptionInvalidArg.exe
2
3  Unhandled Exception:
4  System.ArgumentException: is too much 'a'
5  Parameter name: a
6    at
     <StartupCode$exceptionInvalidArg>.$ExceptionInvalidArg$fsx.main@
     () [0x0000b] in <62028a57941e65a2a7450383578a0262>:0
7  [ERROR] FATAL UNHANDLED EXCEPTION:
     System.ArgumentException: is too much 'a'
8  Parameter name: a
9    at
     <StartupCode$exceptionInvalidArg>.$ExceptionInvalidArg$fsx.main@
     () [0x0000b] in <62028a57941e65a2a7450383578a0262>:0
```

Listing 19.19 exceptionInvalidArgNCatch.fsx:
Catching the exception raised by `invalidArg`.

```
1  let _ =
2    try
3      invalidArg "a" "is too much 'a'"
4    with
5      :? System.ArgumentException -> printfn "Argument is no good!"
```

```
1  $ fsharpc --nologo exceptionInvalidArgNCatch.fsx && mono
      exceptionInvalidArgNCatch.exe
2  Argument is no good!
```

Listing 19.20 exceptionReraise.fsx:
Reraising an exception.

```
1  let _ =
2    try
3      failwith "Arrrrg"
4    with
5      Failure msg ->
6        printfn "The castle of %A" msg
7        reraise ()
```

```
1  $ fsharpc --nologo exceptionReraise.fsx && mono
     exceptionReraise.exe
2  The castle of "Arrrrg"
3
4  Unhandled Exception:
5  System.Exception: Arrrrg
6    at
     <StartupCode$exceptionReraise>.$ExceptionReraise$fsx.main@
     () [0x00041] in <62028a640317d799a7450383648a0262>:0
7  [ERROR] FATAL UNHANDLED EXCEPTION: System.Exception: Arrrrg
8    at
     <StartupCode$exceptionReraise>.$ExceptionReraise$fsx.main@
     () [0x00041] in <62028a640317d799a7450383648a0262>:0
```

19.2 Option Types

At exceptions, it is not always obvious what should be returned. E.g., in the **??**, the exception is handled gracefully, but the return value is somewhat arbitrarily chosen to be the largest possible integer. Instead, we may use the *option type*. The option type is a wrapper that can be put around any type, and which extends the type with the special value *None*. All other values are preceded by the *Some* identifier. An example of rewriting **??** to correctly represent the non-computable value is shown in **??**. The value of an option type can be extracted and tested for by its member

Listing 19.21: Option types can be used when the value in case of exceptions is unclear.

```
1  > let div enum denom =
2  -    try
3  -       Some (enum / denom)
4  -    with
5  -       | :? System.DivideByZeroException -> None;;
6  val div : enum:int -> denom:int -> int option
7
8  >
9  - let a = div 3 1;;
10 val a : int option = Some 3
11
12 > let b = div 3 0;;
13 val b : int option = None
```

functions, *IsNone*, *IsSome*, and *Value*, as illustrated in **??**. The `Value` member

Listing 19.22 option.fsx:
Simple operations on option types.

```
1  let a = Some 3;
2  let b = None;
3  printfn "%A %A" a b
4  printfn "%A %b %b" a.Value b.IsSome b.IsNone
```

```
1  $ fsharpc --nologo option.fsx && mono option.exe
2  Some 3 None
3  3 false true
```

is not defined for `None`, thus it is adviced to **prefer explicit pattern matching for extracting values from an option type.** An example is: `let get (opt :`

'a option) (def : 'a) = match opt with Some x -> x | _ -> def. Note also that printf prints the value None as <null>. This author hopes that future versions of the option type will have better visual representations of the None value.

Functions on option types are defined using the *option*–keyword. E.g., to define a function with explicit type annotation that always returns None, write let f (x : 'a option) = None.

F# includes an extensive Option module. It defines, among many other functions, *Option.bind* which implements let bind f opt = match opt with None -> None | Some x -> f x. The function Option.bind is demonstrated in **??**. The

---

Listing 19.23: Using `Option.bind` to perform calculations on option types.

```
1  > Option.bind (fun x -> Some (2*x)) (Some 3);;
2  val it : int option = Some 6
```

Option.bind is a useful tool for cascading functions that evaluates to option types.

19.3 Programming Intermezzo: Sequential Division of Floats

The following problem illustrates cascading error handling:

Task 19.1

Given a list of floats such as [1.0; 2.0; 3.0], calculate the sequential division 1.0/2.0/3.0.

A sequential division is safe if the list does not contain zero values. However, if any element in the list is zero, then error handling must be performed. An example using failwith is given in **??**. In this example, a recursive function is defined which updates an accumulator element, initially set to the neutral value 1.0. Division by zero results in a failwith exception, wherefore we must wrap its use in a try–with expression.

Instead of using exceptions, we may use Option.bind. In order to use Option.bind for a sequence of non-option floats, we will define a division operator that reverses the order of operands. This is shown in **??**. Here the function divideBy takes two non-option arguments and returns an option type. Thus, Option.bind (divideBy

Listing 19.24 seqDiv.fsx:
Sequentially dividing a list of numbers.

```
1  let rec seqDiv acc lst =
2    match lst with
3      | [] -> acc
4      | elm::rest when elm <> 0.0 -> seqDiv (acc/elm) rest
5      | _ -> failwith "Division by zero"
6
7  try
8    printfn "%A" (seqDiv 1.0 [1.0; 2.0; 3.0])
9    printfn "%A" (seqDiv 1.0 [1.0; 0.0; 3.0])
10 with
11   Failure msg -> printfn "%s" msg
```

```
1  $ fsharpc --nologo seqDiv.fsx && mono seqDiv.exe
2  0.1666666667
3  Division by zero
```

`2.0) (Some 1.0)` is equal to `Some 0.5`, since `divideBy 2.0` is a function that divides any float argument by 2.0. Iterating `Option.bind (divideBy 3.0) (Some 0.5)`, we calculate `Some 0.1666666667` or `Some (1.0/6.0)`, as expected. In **??**, this is written as a single `let`-binding using piping. Since `Option.bind` correctly handles the distinction between `Some` and `None` values, such piping sequences correctly handle possible errors, as shown in **??**.

The sequential application can be extended to lists, using `List.foldBack`, as demonstrated in **??**. Since `List.foldBack` processes the list from the right, the list of integers has been reversed. Notice how `divideByOption` is the function spelled out in each piping step of **??**.

Exceptions and option type are systems to communicate errors up through a hierarchy of function calls. While exceptions favor imperative style programming, option types belong to functional style programming. Exceptions allow for a detailed report of the type of error to the caller, whereas option types only allow for flagging that an error has occurred.

Listing 19.25 seqDivOption.fsx:
Sequentially dividing a sequence of numbers using `Option.bind`. Compare with ??.

```
let divideBy denom enum =
  if denom = 0.0 then
    None
  else
    Some (enum/denom)

let success =
  Some 1.0
  |> Option.bind (divideBy 2.0)
  |> Option.bind (divideBy 3.0)
printfn "%A" success

let fail =
  Some 1.0
  |> Option.bind (divideBy 0.0)
  |> Option.bind (divideBy 3.0)
printfn "%A; isNone: %b" fail fail.IsNone
```

```
$ fsharpc --nologo seqDivOption.fsx && mono
    seqDivOption.exe
Some 0.1666666667
None; isNone: true
```

Listing 19.26 seqDivOptionAdv.fsx:
Sequentially dividing a list of numbers, using `Option.bind` and `List.foldBack`. Compare with ??.

```
let divideBy denom enum =
  if denom = 0.0 then
    None
  else
    Some (enum/denom)
let divideByOption x acc =
  Option.bind (divideBy x) acc

let success = List.foldBack divideByOption [3.0; 2.0; 1.0] (Some
    1.0)
printfn "%A" success

let fail = List.foldBack divideByOption [3.0; 0.0; 1.0] (Some 1.0)
printfn "%A; isNone: %A" fail fail.IsNone
```

```
$ fsharpc --nologo seqDivOptionAdv.fsx && mono
    seqDivOptionAdv.exe
Some 0.1666666667
None; isNone: true
```

20 Working With Files

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. Graphical user interfaces will be discussed in **??**, and 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, the `printf` family of functions discussed in Section 2.3 is 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.

20.1 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 20.1: Defining an entry point function with arguments from the console.

```
1  [<EntryPoint>]
2  let <funcIdent> <arg> =
3    <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 **??**. An example execution with arguments is shown in **??**. In Bash, the return value is

Listing 20.2 commandLineArgs.fsx:
Interacting with a user with `ReadLine` and `WriteLine`.

```
1  [<EntryPoint>]
2  let main args =
3      printfn "Arguments passed to function : %A" args
4      0  // Signals that program terminated successfully
```

called the *exit status* and can be tested using Bash's `if` statements, as demonstrated in **??**. Also in Bash, the exit status of the last executed program can be accessed using the `$?` built-in environment variable. In Windows, this same variable is called `%errorlevel%`.

Listing 20.3: An example dialogue of running ??.

```
1  $ fsharpc --nologo commandLineArgs.fsx
2  $ mono commandLineArgs.exe Hello World
3  Arguments passed to function : [|"Hello"; "World"|]
```

Listing 20.4: Testing return values in Bash when running ??.

```
1  $ fsharpc --nologo commandLineArgs.fsx
2  $ if mono commandLineArgs.exe Hello World; then echo
     "success"; else echo "failure"; fi
3  Arguments passed to function : [|"Hello"; "World"|]
4  success
```

20.2 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 ??. On the console, the standard output and error streams

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 20.1: Three built-in streams in `System.Console`.

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 ??. Note that you must supply the empty argument "()" to the `Read` functions in order to run most of the functions instead of referring to them as

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 20.2: Some functions for interacting with the user through the console in the System.Console class. Prefix "`System.Console.`" is omitted for brevity.

values. A demonstration of the use of `Write`, `WriteLine`, and `ReadLine` is given in **??**. The functions `Write` and `WriteLine` act as `printfn` without a formatting string. These functions have 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 20.5 userDialogue.fsx:
Interacting with a user with `ReadLine` and `WriteLine`. The user typed
"3.5" and "7.4".

```
1  System.Console.WriteLine "To perform the multiplication of a and
      b"
2  System.Console.Write "Enter a: "
3  let a = float (System.Console.ReadLine ())
4  System.Console.Write "Enter b: "
5  let b = float (System.Console.ReadLine ())
6  System.Console.WriteLine ("a * b = " + string (a * b))
```

```
1  $ fsharpc --nologo userDialogue.fsx && mono
      userDialogue.exe
2  To perform the multiplication of a and b
3  Enter a: 3.5
4  Enter b: 7.4
5  a * b = 25.9
```

20.3 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
will protect it depending on how the file is opened. E.g., if you are going to write to
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

Listing 20.6 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.

```fsharp
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 ())
```

```
$ fsharpc --nologo readKey.fsx && mono readKey.exe
Start typing
You pressed: A
You pressed: SHIFT+A
You pressed: CTRL+A
```

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 **??**.

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 **??**. An example of how a file is opened and later closed is shown in **??**. 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.

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 **??**, will result in an `FileStream` object. From this object we can extract infor-

System.IO.File	Description
Open: (path : string) * (mode : FileM -> FileStream	Request the opening of a file on `path` for reading and writing with access mode `FileMode`, see **??**. 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 20.3: The family of `System.IO.File.Open` functions. See **??** for a description of `FileMode`, **????** for a description of `FileStream`, **??** for a description of `StreamReader`, and **??** for a description of `StreamWriter`.

mation about the file, such as the permitted operations and more listed in **??**. This information is important in order to restrict the operation that we will perform on the file. Some typical operations are listed in and **??**. 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.

Text is typically streamed through the `StreamReader` and `StreamWriter`. These may be considered higher-order stream processing, since they include an added interpretation of the bits to strings. A `StreamReader` has methods similar to a `FileStream`

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 20.4: File mode values for the `System.IO.Open` 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 20.5: Some properties of the `System.IO.FileStream` class.

object and a few new properties and methods, such as the `EndOfStream` property and `ReadToEnd` method, see **??**. Likewise, a `StreamWriter` has an added method for automatically flushing after every writing operation. A simple example of opening a text-file and processing it is given in **??**. Here the program reads the source code of itself, and prints it to the console.

20.4 Working With Files and Directories.

F# has support for managing files, summarized in the `System.IO.File` class and summarized in **??**.

Listing 20.7 openFile.fsx:
Opening and closing a file, in this case, the source code of this same file.

```
1  let filename = "openFile.fsx"
2
3  let reader =
4    try
5      Some (System.IO.File.Open (filename, System.IO.FileMode.Open))
6    with
7      _ -> None
8
9  if reader.IsSome then
10    printfn "The file %A was successfully opened." filename
11   reader.Value.Close ()
```

```
1  $ fsharpc --nologo openFile.fsx && mono openFile.exe
2  The file "openFile.fsx" was successfully opened.
```

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 20.6: Some methods of the `System.IO.FileStream` class.

In the `System.IO.Directory` class there are a number of other frequently used functions, summarized in **??**.

> **Listing 20.8 readFile.fsx:**
> **An example of opening a text file and using the `StreamReader` properties and methods.**
>
> ```fsharp
> let printFile (reader : System.IO.StreamReader) =
>   while not(reader.EndOfStream) do
>     let line = reader.ReadLine ()
>     printfn "%s" line
>
> let filename = "readFile.fsx"
> let reader = System.IO.File.OpenText filename
> printFile reader
> ```
> ```
> $ fsharpc --nologo readFile.fsx && mono readFile.exe
> let printFile (reader : System.IO.StreamReader) =
>   while not(reader.EndOfStream) do
>     let line = reader.ReadLine ()
>     printfn "%s" line
>
> let filename = "readFile.fsx"
> let reader = System.IO.File.OpenText filename
> printFile reader
> ```

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 20.7: Some methods of the `System.IO.StreamReader` class.

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 20.8: Some methods of the `System.IO.StreamWriter` class.

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 20.9: Some methods of the `System.IO.File` class.

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 20.10: 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 **??**.

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 20.11: Some methods of the `System.IO.Path` class.

20.5 Programming intermezzo: Name of Existing File Dialogue

A typical problem when working with files is

Task 20.1

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 **??**. Note that

---

Listing 20.9 filenamedialogue.fsx:
Ask the user to input a name of an existing file.

```
1  let getAFileName () =
2    let mutable filename = ""
3    let mutable fileExists = false
4    while not(fileExists) do
5      System.Console.Write("Enter Filename: ")
6      filename <- System.Console.ReadLine()
7      fileExists <- System.IO.File.Exists filename
8    filename
9
10 let listOfFiles = System.IO.Directory.GetFiles "."
11 printfn "Directory contains: %A" listOfFiles
12 let filename = getAFileName ()
13 printfn "You typed: %s" filename
```

---

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.

20.6 Reading From the Internet

The internet is a global collection of computers that are connected in a network using the internet protocol suite TCP/IP. The internet is commonly used for transport of data such as emails and for offering services such as web pages on the World Wide Web. Web resources are identified by a *Uniform Resource Locator* (*URL*), popularly known as a web page, and an URL contains information about where and how data from the web page is to be obtained. E.g.,

```
https://en.wikipedia.org/wiki/F_Sharp_(programming_language)
```

contains 3 pieces of information: The host uses the `https` protocol, `en.wikipedia.org` is its name, and `wiki/F_Sharp_(programming_language)` is the filename.

F#'s `System` namespace contains functions for accessing web pages as stream, as illustrated in **??**. To connect to a URL as a stream, we first need first format the URL string as a *Uniform Resource Identifiers* (*URI*), which is a generalization of the

> **Listing 20.10 webRequest.fsx:**
> **Downloading a web page and printing the first few characters.**

```fsharp
/// Set up a url as a stream
let url2Stream url =
    let uri = System.Uri url
    let request = System.Net.WebRequest.Create uri
    let response = request.GetResponse ()
    response.GetResponseStream ()

/// Read all contents of a web page as a string
let readUrl url =
    let stream = url2Stream url
    let reader = new System.IO.StreamReader(stream)
    reader.ReadToEnd ()

let url = "http://fsharp.org"
let a = 40

let html = readUrl url
printfn "Downloaded %A. First %d characters are: %A" url a
    html.[0..a]
```

```
$ fsharpc --nologo webRequest.fsx && mono webRequest.exe
Downloaded "http://fsharp.org". First 40 characters are:
    "<!DOCTYPE html>
<html lang="en">
<head>"
```

URL concept, using the `System.Uri` function. Then we must initialize the request by the `System.Net.WebRequest` function, and the response from the host is obtained by the `GetResponse` method. Finally, we can access the response as a stream by the `GetResponseStream` method. In the end, we convert the stream to a `StreamReader`, such that we can use the methods from **??** to access the web page.

20.7 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 **??** for more on interfaces.

The *use* keyword is similar to `let`:

Listing 20.11:   Use binding expression.

```
1   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 **??**.   Here, `file` is an `System.IDisposable` object, and `file.Dispose()`

Listing 20.12 useBinding.fsx:
Using use instead of let releases disposable resources at end of scope.

```
1   open System.IO
2
3   let writeToFile (filename : string) (str : string) : unit =
4       use file = File.CreateText filename
5       file.Write str
6       // file.Dispose() is implicitly called here,
7       // implying that the file is closed.
8
9   writeToFile "use.txt" "Using 'use' closes the file, when out of
        scope."
```

---

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 **??** 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.**

Listing 20.13 using.fsx:
The using function executes a function on an object and releases its disposable resources. Compare with ??.

```
1  open System.IO
2
3  let writeToFile (str : string) (file : StreamWriter) : unit =
4      file.Write str
5
6  using (File.CreateText "use.txt") (writeToFile "Disposed after
       call.")
7  // Dispose() is implicitly called on the anonymous file handle,
       implying
8  // that the file is automatically closed.
```

21 Classes and 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 **??**. In

aClass
// The object's values (properties)
aValue : int
anotherValue : float*bool
// The object's functions (methods)
aMethod: () -> int
anotherMethod: float -> float

Figure 21.1: A class is sometimes drawn as a figure.

this illustration, objects of type `aClass` will each contain an int and a pair of a 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 **??**.

21.1 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 21.1:   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 **??**. Here we have defined a class `car`, instantiated three objects, and bound them to the names `sportsCar`, `stationWagon`, and `miniBus`.

car
// A car has a color color : string

sportsCar: car		stationWagon: car		miniBus: car
color = "red"		color = "green"		color = "blue"
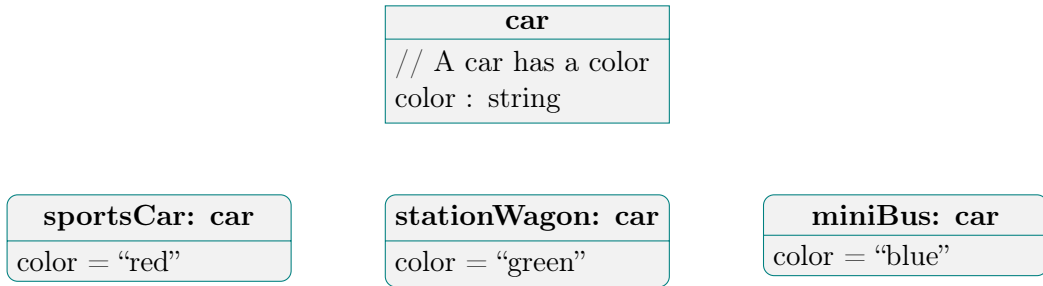
Figure 21.2: A class `car` is instantiated trice and bound to the names `sportsCar`, `stationWagon`, and `miniBus`, and each object's properties are set to different values.

Each object has been given different values for the `color` property. In F# this could look like the code in **??**. In the example, the class `car` is defined in lines **??**–**??**. Its

---

Listing 21.2 car.fsx:
Defining a class car, and making three instances of it. See also ??.

```
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
```

```
1  $ fsharpc --nologo car.fsx && mono car.exe
2  red green blue
```

---

header includes one string argument, `aColor`. The body of the constructor is empty, and the member section consists of lines **??**–**??**. 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 **??**–**??**, and the value of their properties are accessed in line **??**. 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 **??**. Here in line **??**, the constructor has 2

Listing 21.3 class.fsx:
Extending ?? with fields and methods.

```
1   type car (econ : float, fuel : float) =
2     // Constructor body section
3     let mutable fuelLeft = fuel // liters in the tank
4     do printfn "Created a car (%.1f, %.1f)" econ fuel
5     // Member section
6     member this.fuel = fuelLeft
7     member this.drive distance =
8       fuelLeft <- fuelLeft - econ * distance / 100.0
9
10  let sport = car (8.0, 60.0)
11  let economy = car (5.0, 45.0)
12  sport.drive 100.0
13  economy.drive 100.0
14  printfn "Fuel left after 100km driving:"
15  printfn " sport: %.1f" sport.fuel
16  printfn " economy: %.1f" economy.fuel
```

```
1   $ fsharpc --nologo class.fsx && mono class.exe
2   Created a car (8.0, 60.0)
3   Created a car (5.0, 45.0)
4   Fuel left after 100km driving:
5    sport: 52.0
6    economy: 40.0
```

---

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 **??**–**??**. 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

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*.

21.2 Accessors

Methods are most often used as an interface between the fields of an object and the application program. Consider the example in **??**. In the example, the data

> **Listing 21.4 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 ())
> ```
> ```
> $ fsharpc --nologo classAccessor.fsx && mono
>     classAccessor.exe
> 1
> 2
> ```

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 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 **??**. The expression defining `get: () -> 'a`

Listing 21.5 classGetSet.fsx:
Members can act as variables with the built-in get and set functions.

```
1  type aClass () =
2    let mutable v = 0
3    member this.value
4      with get () = v
5      and set (a) = v <- a
6
7  let a = aClass ()
8  printfn "%d" a.value
9  a.value<-2
10 printfn "%d" a.value
```

```
1  $ fsharpc --nologo classGetSet.fsx && mono classGetSet.exe
2  0
3  2
```

---

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 **??**. Higher dimensional indexed properties are defined by adding more indexing arguments to the definition of `get` and `set`, such as demonstrated in **??**.

Listing 21.6 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]
```

```
$ fsharpc --nologo classGetSetIndexed.fsx && mono
    classGetSetIndexed.exe
ClassGetSetIndexed+aClass
0 0 0
0 3 0
```

Listing 21.7 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]
```

```
$ fsharpc --nologo classGetSetHigherIndexed.fsx && mono
    classGetSetHigherIndexed.exe
ClassGetSetHigherIndexed+aClass
0 0 0
0 3 0
```

21.3 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*, see **??**. Consider the example in **??**. Thus, the binding to b in line **??** 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 **??**. 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 21.8 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  $ fsharpc --nologo classReference.fsx && mono
     classReference.exe
2  2 2
```

21.4 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 **??**. Notice in line **??** that a static field nextAvailableID is created for the value

Listing 21.9 classCopy.fsx:
A copy method is often needed. Compare with ??.

```
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  $ fsharpc --nologo classCopy.fsx && mono classCopy.exe
2  2 0
```

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 **??**, and `nextAvailableID` is updated. The static field can be accessed with a static accessor, as demonstrated in line **??**. Notice how this definition does not include a self-identifier, and that the member is accessible from the application in line **??** using the class' name, in both cases since it is not a member of any particular object.

21.5 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 **??**. For mutually recursive classes, the keyword *and* must be used, as shown in **??**. Here `anInt` and `aFloat` 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`.

Listing 21.10 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  $ fsharpc --nologo classStatic.fsx && mono classStatic.exe
2  Jon: 0,  Hans: 1
3  Next id: 2
```

21.6 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 **??**. In the example we define an object which can produce greetings 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 **??**, the notation for addition is less than elegant. For such situations, F# supports *operator overloading*. All usual operators may be overloaded (see **??**), and in contrast to regular operator overloading, 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 **??**. 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 by its function-equivalent. Note that the functional equivalence of the

> **Listing 21.11 classRecursion.fsx:**
> **Members can recurse without the** `rec` **keyword and refer to other members regardless of their lexicographical scope.**

```fsharp
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
```

```
$ fsharpc --nologo classRecursion.fsx && mono
    classRecursion.exe
4 4 6
```

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 **??**, 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.**

> **Listing 21.12 classAssymetry.fsx:**
> Mutually recursive classes are defined using the `and` keyword.

```fsharp
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
```

```
$ fsharpc --nologo classAssymetry.fsx && mono
  classAssymetry.exe
2 3.2 5.2 5.2
```

21.7 Additional Constructors

Like methods, constructors can also be overloaded by using the `new` keyword. E.g., the example in **??** 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 **??**. The top constructor that does not use the `new`-keyword is called the *primary 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 **??**. The `do`-keyword is often understood to be

Listing 21.13 classOverload.fsx:
**Overloading methods set : int -> () and set : int * int -> () is
permitted, since they differ in argument number or type.**

```fsharp
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
```

```
$ fsharpc --nologo classOverload.fsx && mono
    classOverload.exe
Hi Programmer
Hi F# programmer
Hello Expert
```

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.

Listing 21.14 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
```

```
$ fsharpc --nologo classOverloadOperator.fsx && mono
  classOverloadOperator.exe
a=2, b=3, c=3.2, d=5, e=5.2, f=5.2, g=-2, h=-1
```

Listing 21.15 classExtraConstructor.fsx:
Extra constructors can be added, using `new`.

```
1  type classExtraConstructor (name : string, greetings : string) =
2    static let defaultGreetings = "Hello"
3   // Additional constructors are defined by new ()
4    new (name : string) =
5      classExtraConstructor (name, defaultGreetings)
6    member this.name = name
7    member this.str = greetings + " " + name
8
9  let s = classExtraConstructor ("F#") // Calling additional
      constructor
10  let t = classExtraConstructor ("F#", "Hi") // Calling primary
      constructor
11  printfn "%A, %A" s.str t.str
```

```
1  $ fsharpc --nologo classExtraConstructor.fsx && mono
      classExtraConstructor.exe
2  "Hello F#", "Hi F#"
```

Listing 21.16 classDoThen.fsx:
The optional do- and then-keywords allow for computations before and after the primary constructor is called.

```
1   type classDoThen (aValue : float) =
2     // "do" is mandatory to execute code in the primary constructor
3     do printfn "  Primary constructor called"
4     // Some calculations
5     do printfn "  Primary done" (* *)
6     new () =
7       // "do" is optional in additional constructors
8       printfn "  Additional constructor called"
9       classDoThen (0.0)
10      // Use "then" to execute code after construction
11      then
12        printfn "  Additional done"
13    member this.value = aValue
14
15  printfn "Calling additional constructor"
16  let v = classDoThen ()
17  printfn "Calling primary constructor"
18  let w = classDoThen (1.0)
```

```
1   $ fsharpc --nologo classDoThen.fsx && mono classDoThen.exe
2   Calling additional constructor
3     Additional constructor called
4     Primary constructor called
5     Primary done
6     Additional done
7   Calling primary constructor
8     Primary constructor called
9     Primary done
```

21.8 Programming Intermezzo: Two Dimensional Vectors

Consider the following problem.

Task 21.1

A Euclidean vector is a geometric object that has a direction, a length, and two operations: vector addition and scalar multiplication, see **??**. Define a class for a vector in two dimensions.
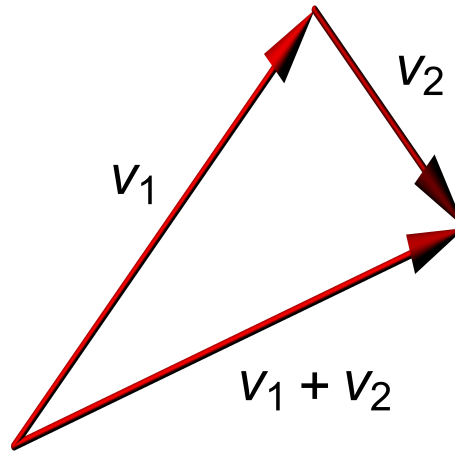
Figure 21.3: Illustration of vector addition in two dimensions.

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,

$$\vec{v} = (x, y), \tag{21.7}$$

the basic operations are defined as

$$\vec{v}_1 + \vec{v}_2 = (x_1 + x_2, y_1 + y_2), \tag{21.8}$$

$$a\vec{v} = (ax, ax), \tag{21.9}$$

$$\mathrm{dir}(\vec{v}) = \tan \frac{y}{x},\ x \neq 0, \tag{21.10}$$

$$\mathrm{len}(\vec{v}) = \sqrt{x^2 + y^2}, \tag{21.11}$$

where x_i and y_i are the elements of vector $\vec{v}_i$, 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 $0 \le \theta < 2\pi$ and $0 \le l$. This representation reminds us that vectors do not have a position. For vectors on polar form,

$$\vec{v} = (\theta, l), \tag{21.12}$$

their basic operations are defined as

$$x(\theta, l) = l \cos(\theta), \tag{21.13}$$
$$y(\theta, l) = l \sin(\theta), \tag{21.14}$$
$$\vec{v}_1 + \vec{v}_2 = (x(\theta_1, l_1) + x(\theta_2, l_2), y(\theta_1, l_1) + y(\theta_2, l_2)) \tag{21.15}$$
$$a\vec{v} = (\theta, al), \tag{21.16}$$

where θ_i and l_i are the elements of vector $\vec{v}_i$, 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 **??**. 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`

Listing 21.17 vectorApp.fsx:
An application using the library in ??.

```
1  open Geometry
2  let v = vector(Cartesian (1.0,2.0))
3  let w = vector(Polar (3.2,1.8))
4  let p = vector()
5  let q = vector(1.2, -0.9)
6  let a = 1.5
7  printfn "%A * %A = %A" a v (a * v)
8  printfn "%A + %A = %A" v w (v + w)
9  printfn "vector() = %A" p
10 printfn "vector(1.2, -0.9) = %A" q
11 printfn "v.dir = %A" v.dir
12 printfn "v.len = %A" v.len
```

family. Thus, we have further sketched requirements to the library with the emphasis on application.

After a couple of trials, our library implementation has ended up as shown in **??**.
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 **??**. The output is as expected, and for the vector class, our solution seems to be a good compromise between versatility and syntactical bloating.

Listing 21.18 vector.fs:
A library serving the application in ??.

```fsharp
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 21.19: Compiling and running the code from ?? and ??.

```
$ fsharpc --nologo vector.fs vectorApp.fsx && mono
   vectorApp.exe
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
```

22 Derived Classes

22.1 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 **??** and is called an *is-a relation*. Is-a relations can be implemented using class *inheritance*,
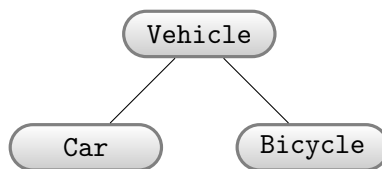
Figure 22.1: Both a car and a bicycle is a (type of) vehicle.

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 **??** is:

Listing 22.1: A class definition with inheritance.

```
1  type <classIdent> ({<arg>}) [as <selfIdent>]
2     [inherit <baseClassIdent>({<arg>})]
3     {[let <binding>] | [do <statement>]}
4     {(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 **??**. In

Listing 22.2 vehicle.fsx:
New classes can be derived from old ones.

```
1  /// All vehicles have wheels
2  type vehicle (nWheels : int ) =
3    member this.wheels = nWheels
4
5  /// A car is a vehicle with 4 wheels
6  type car (nPassengers : int) =
7    inherit vehicle (4)
8    member this.maxPassengers = nPassengers
9
10 /// A bicycle is a vehicle with 2 wheels
11 type bicycle () =
12   inherit vehicle (2)
13   member this.mustUseHelmet = true
14
15 let aVehicle = vehicle (1)
16 let aCar = car (4)
17 let aBike = bicycle ()
18 printfn "aVehicle has %d wheel(s)" aVehicle.wheels
19 printfn "aCar has %d wheel(s) with room for %d passenger(s)"
     aCar.wheels aCar.maxPassengers
20 printfn "aBike has %d wheel(s). Is helmet required? %b"
     aBike.wheels aBike.mustUseHelmet
```

```
1  $ fsharpc --nologo vehicle.fsx && mono vehicle.exe
2  aVehicle has 1 wheel(s)
3  aCar has 4 wheel(s) with room for 4 passenger(s)
4  aBike has 2 wheel(s). Is helmet required? true
```

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.

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 **??**.   In this case, we have defined three greetings:

> **Listing 22.3 memberOvershadowingVar.fsx:**
> **Inherited members can be overshadowed, but we can still access the base member. Compare with ??.**

```
1  /// hi is a greeting
2  type greeting () =
3    member this.str = "hi"
4  /// hello is a greeting
5  type hello () =
6    inherit greeting ()
7    member this.str = "hello"
8  /// howdy is a greeting
9  type howdy () =
10   inherit greeting ()
11   member this.str = "howdy"
12
13 let a = greeting ()
14 let b = hello ()
15 let c = howdy ()
16 printfn "%s, %s, %s" a.str b.str c.str
```

```
1  $ fsharpc --nologo memberOvershadowingVar.fsx && mono
     memberOvershadowingVar.exe
2  hi, hello, howdy
```

`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 **??**.   Here `howdy` is derived from `hello`, overshadows `str`, and adds property `altStr`. By upcasting object `b`, we create object `c` as a copy of

Listing 22.4 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.

```
1   /// hello holds property str
2   type hello () =
3     member this.str = "hello"
4   /// howdy is a hello class and has property altStr
5   type howdy () =
6     inherit hello ()
7     member this.str = "howdy"
8     member this.altStr = "hi"
9
10  let a = hello ()
11  let b = howdy ()
12  let c = b :> hello // a howdy object as if it were a hello object
13  printfn "%s %s %s %s" a.str b.str b.altStr c.str
```

```
1   $ fsharpc --nologo upCasting.fsx && mono upCasting.exe
2   hello howdy hi hello
```

`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* operator `:?>`, but the validity of the operation is checked at runtime. Thus, **avoid downcasting when possible.**

22.2 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 **??**.   All classes

---

Listing 22.5 classPrintf.fsx:
Printing classes yields low-level information about the class.

```
1  type vectorDefaultToString (x : float, y : float) =
2    member this.x = (x,y)
3
4  let v = vectorDefaultToString (1.0, 2.0)
5  printfn "%A" v // Printing objects gives low-level information
```

```
1  $ fsharpc --nologo classPrintf.fsx && mono classPrintf.exe
2  ClassPrintf+vectorDefaultToString
```

---

implicitly 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 **??**. Note, despite that `ToString()` returns a string, the `%s` placerholder only accepts values of the basic `string` type.   We see 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.

> **Listing 22.6 classToString.fsx:**
> Overriding `ToString()` function for better interaction with members of the `printf` family of procedures. Compare with **??**.

```
1  type vectorWToString (x : float, y : float) =
2    member this.x = (x,y)
3    // Custom printing of objects by overriding this.ToString()
4    override this.ToString() =
5      sprintf "(%A, %A)" (fst this.x) (snd this.x)
6
7  let v = vectorWToString(1.0, 2.0)
8  printfn "%A" v // No change in application but result is better
```

```
1  $ fsharpc --nologo classToString.fsx && mono
     classToString.exe
2  (1.0, 2.0)
```

22.3 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 **??**. 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

> **Listing 22.7 abstractClass.fsx:**
> **In contrast to regular objects, upcasted derived objects use the derived implementation of abstract methods. Compare with ??.**

```
1  /// An abstract class for general greeting classes with property
     str
2  [<AbstractClass >]
3  type greeting () =
4    abstract member str : string
5  /// hello is a greeting
6  type hello () =
7    inherit greeting ()
8    override this.str = "hello"
9  /// howdy is a greeting
10 type howdy () =
11   inherit greeting ()
12   override this.str = "howdy"
13
14 let a = hello ()
15 let b = howdy ()
16 let c = [| a :> greeting; b :> greeting |] // arrays of greetings
17 Array.iter (fun (elm : greeting) -> printfn "%s" elm.str) c
```

```
1  $ fsharpc --nologo abstractClass.fsx && mono
     abstractClass.exe
2  hello
3  howdy
```

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 of implementations being available in the abstract class, the abstract class still cannot be used to instantiate objects. The example in **??** shows an extension of **??** with a default implementation. In the example, the program in **??** 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

---

Listing 22.8 abstractDefaultClass.fsx:
Default implementations in abstract classes make implementations in
derived classes optional. Compare with ??.

```
1  /// An abstract class for general greeting classes with property
       str
2  [<AbstractClass >]
3  type greeting () =
4    abstract member str : string
5    default this.str = "hello" // Provide default implementation
6  /// hello is a greeting
7  type hello () =
8    inherit greeting ()
9  /// howdy is a greeting
10 type howdy () =
11   inherit greeting ()
12   override this.str = "howdy"
13
14 let a = hello ()
15 let b = howdy ()
16 let c = [| a :> greeting; b :> greeting |] // arrays of greetings
17 Array.iter (fun (elm : greeting) -> printfn "%s" elm.str) c
```

```
1  $ fsharpc --nologo abstractDefaultClass.fsx && mono
       abstractDefaultClass.exe
2  hello
3  howdy
```

is an abstract class defining among other members, the `ToString` method with default
implementation.

22.4 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 **??**. 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 **??**, the `printfn` function only needs to know the member's type, not its meaning. As a consequence, 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.

Listing 22.9 classInterface.fsx:
Interfaces specify which members classes contain, and with upcasting gives more flexibility than abstract classes.

```fsharp
/// 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
```

```
$ fsharpc --nologo classInterface.fsx && mono
    classInterface.exe
value = 2, fct(80.0) = 11200
value = 6, fct(80.0) = 24.6914
```

22.5 Programming Intermezzo: Chess

To demonstrate the use of hierarchies, consider the following problem.

> **Task 22.1**
>
> The 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 **??**.
>
> 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.
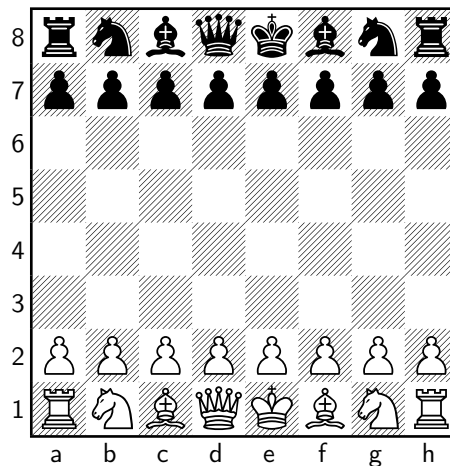
Figure 22.2: Starting position for the game of chess.

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 **??**, and an illustration of what the program should do is shown in **??**. 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 **??**, pieces are upcasted to `chessPiece`, thus,

Listing 22.10 chessApp.fsx:
A chess application.

```fsharp
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
```
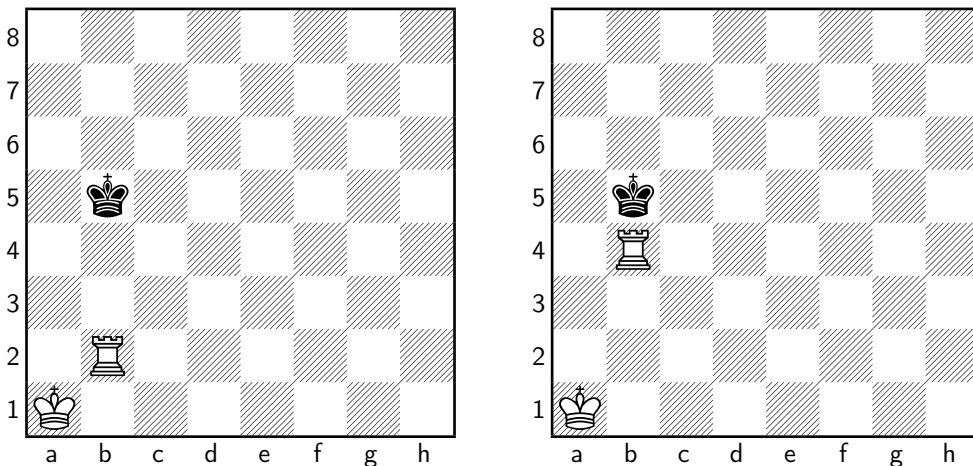

Figure 22.3: Starting at the left and moving white rook to b4.

the base class must know how to print the piece type. For this, we will define an
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 22.11 pieces.fs:
An extension of chess base.

```
1   module Pieces
2   open Chess
3   /// A king moves 1 square in any direction
4   type king(col : Color) =
5     inherit chessPiece(col)
6     // A king has runs of length 1 in 8 directions:
7     //   (N, NE, E, SE, S, SW, W, NW)
8     override this.candidateRelativeMoves =
9         [[(-1,0)];[(-1,1)];[(0,1)];[(1,1)];
10        [(1,0)];[(1,-1)];[(0,-1)];[(-1,-1)]]
11    override this.nameOfType = "king"
12  /// A rook moves horizontally and vertically
13  type rook(col : Color) =
14    inherit chessPiece(col)
15    // A rook can move horizontally and vertically
16    // Make a list of relative coordinate lists. We consider the
17    // current position and try all combinations of relative
18    // moves (1,0); (2,0) ... (7,0); (-1,0); (-2,0); ...; (0,-7).
19    // Some will be out of board, but will be assumed removed as
20    // illegal moves.
21    // A list of functions for relative moves
22    let indToRel = [
23      fun elm -> (elm,0); // South by elm
24      fun elm -> (-elm,0); // North by elm
25      fun elm -> (0,elm); // West by elm
26      fun elm -> (0,-elm) // East by elm
27      ]
28    // For each function f in indToRel, we calculate
29    //   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 **??**. However,

---

Listing 22.12 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
```

---

this solution is imperative in nature and does not use the elegance of the functional programming paradigm. A direct translation into functional programming is given in **??**. The functional version is slightly longer, but avoids the mutable variable.

Listing 22.13 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
```

Generating lists of runs from the two lists `[1..7]` and `indToRel` can also be performed with two `List.map`s, as shown in **??**.

Listing 22.14 ListMapRuns.fsx:
Calculating the runs of a rook using double `List.map`s.

```
30   List.map (fun e -> List.map e [1..7]) indToRel
```

The anonumous function,

```
     fun e -> List.map e [1..7],
```

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 anony-
mous 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 **??** 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 **??**. The `chessPiece` will need to know what a board is, so we must

Listing 22.15 chess.fs:
A chess base: Module header and discriminated union types.

```
1   module Chess
2   type Color = White | Black
3   type Position = int * int
```

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 22.16 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
```

Our `Board` class is by far the largest and will be discussed in **??**–**??**. The constructor is shown in **??**. For memory efficiency, the board has been implemented using a `Array2D`, since pieces will move around often. For later use, in the members shown in **??** 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 **??**. 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 **??**. 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

Listing 22.17 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
```

chess notation.

Listing 22.18 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
??. A chess piece must implement `candidateRelativeMoves`, and we decided in

Listing 22.19 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)
```

?? 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 **??**. We see that the program has correctly determined that initially, the white

---

Listing 22.20: Running the program. Compare with ??.

```
1  $ fsharpc --nologo chess.fs pieces.fs chessApp.fsx && mono
     chessApp.exe
2  7
3  6
4  5
5  4   k
6  3
7  2
8  1   R
9  0 K
10    0 1 2 3 4 5 6 7
11 K: Some (0, 0) ([(0, 1); (1, 0)], [R])
12 R: Some (1, 1) ([(2, 1); (3, 1); (0, 1); (1, 2); (1, 3);
     (1, 4); (1, 5); (1, 6); (1, 7); (1, 0)],
13   [k])
14 k: Some (4, 1) ([(3, 1); (3, 2); (4, 2); (5, 2); (5, 1);
     (5, 0); (4, 0); (3, 0)], [])
15 7
16 6
17 5
18 4   k
19 3   R
20 2
21 1
22 0 K
23    0 1 2 3 4 5 6 7
24 K: Some (0, 0) ([(0, 1); (1, 1); (1, 0)], [])
25 R: Some (3, 1) ([(2, 1); (1, 1); (0, 1); (3, 2); (3, 3);
     (3, 4); (3, 5); (3, 6); (3, 7); (3, 0)],
26   [k])
27 k: Some (4, 1) ([(3, 2); (4, 2); (5, 2); (5, 1); (5, 0);
     (4, 0); (3, 0)], [R])
```

---

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.

23 The Object-Oriented Programming Paradigm

Object-oriented programming is a paradigm for encapsulating data and methods into cohesive units. 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 specialty. 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 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 **??**–**??** are the analysis phase which gradually stops in step **??**, while the design phase gradually starts at step **??** and gradually stops when actual code is written in step **??**. Notice that the last steps are identical to imperative programming, **??**. 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.

23.1 Identification of Objects, Behaviors, and Interactions by Nouns-and-Verbs

A key point in object-oriented programming is that objects should to a large extent be independent and reusable. As an example, the type `int` models the concept of integer numbers. It can hold integer values from -2,147,483,648 to 2,147,483,647, and a number of standard operations and functions are defined for it. We may use integers in many different programs, and it is certain that the original designers did not foresee our use, but strived to make a general type applicable for many uses. Such a design is a useful goal when designing objects, that is, our objects should model the general concepts and be applicable in future uses.

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.

23.2 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*) [?] 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 **??**. In UML, classes are represented as boxes with their

ClassName
value-identifier : type   value-identifier : type = default value
function-identifier (arg : type) (arg : type) ... : type   *function-identifier (arg : type) (arg : type) ... : type*
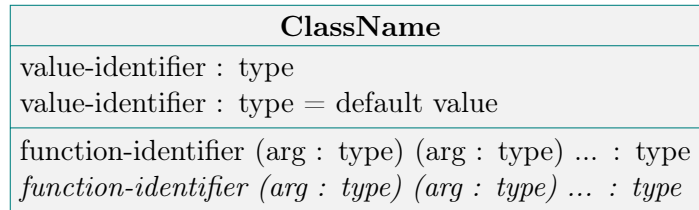
Figure 23.1: A UML diagram for a class consists of it's name, zero or more attributes, and zero or more methods.

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 **??**.

Relations between classes and objects are indicated by lines and arrows. The most common ones are summarized in **??**. Their meaning will be described in detail in the following.

<<interface>> **InterfaceName**		
value-identifier : type value-identifier : type = default value		
function-identifier (arg : type) (arg : type) ... : type		

Figure 23.2: An interface is a class that requires an implementation.
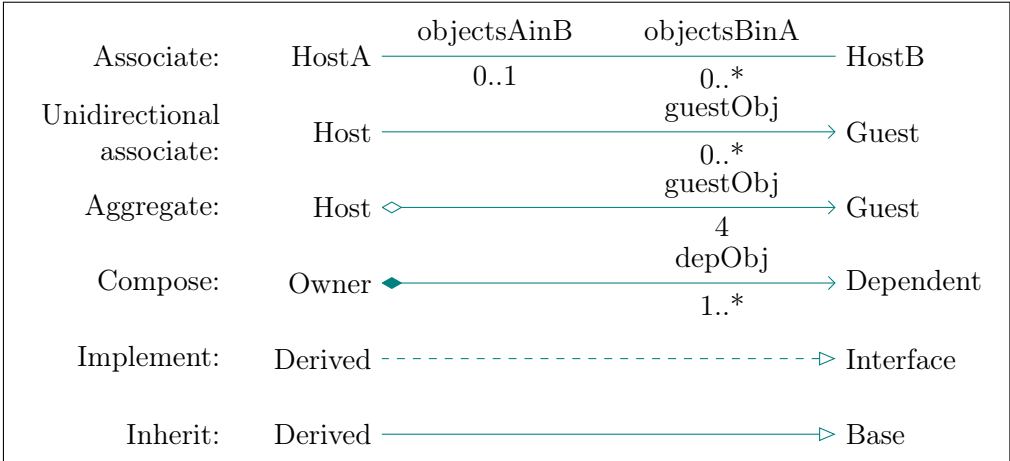
Figure 23.3: Arrows used in class diagrams to show relations between objects.

23.2.1 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 **??**. 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 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*.
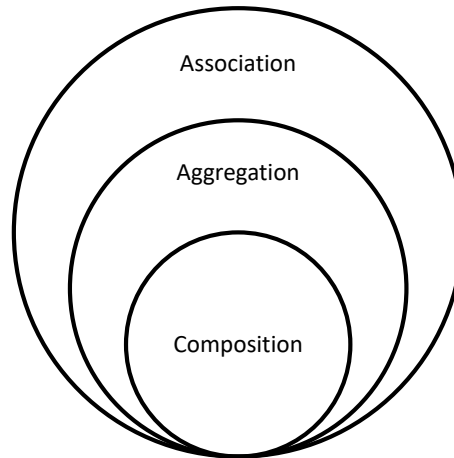
Figure 23.4: The relation between Association, Aggregation and Composition in UML.

Association

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 **??**. Association may be annotated by an identifier and a multiplicity.

Figure 23.5: Bidirectional association is shown as a line with optional annotation.

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 **??**. If the association is unidirectional, then an arrow is added for emphasis, as shown in **??**. In this example, Host knows about Guest and has one instance of it, and Guest is oblivious about Host.

A programming example showing a unidirectional association is given in **??**. Here, the `student` is unidirectionally associated with a `teacher` since the `student` can send

n	exactly n instances
*	zero or more instances
n..m	n to m instances
n..*	from n to infinite instances

Table 23.1: Notation for association multiplicities is similar to F#'s slicing notation.

Figure 23.6: Unidirectional association shows a one-side *has-a* relation.

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 23.7: The `teacher` and `student` objects can access each other's functions, and thus they have an association relation.

Aggregation

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 **??**.

Listing 23.1 umlAssociation.fsx:
The `student` is associated with a `teacher`.

```
1  type teacher () =
2    member this.answer (q : string) = "4"
3  type student (t : teacher) =
4    member this.ask () = t.answer("What is 2+2?")
5
6  let t = teacher ()
7  let s = student (t)
8  s.ask ()
```

Here the Host class has stored aliases to four different Guest objects.

Figure 23.8: 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 **??**.   In aggregated

```
Listing 23.2 umlAggregation.fsx:
The book has an aggregated relation to author and reader.

1   type book (name : string) =
2     let mutable _name = name
3   type author () =
4     let _book = book("Learning to program")
5     member this.publish() = _book
6   type reader () =
7     let mutable _book : book option = None
8     member this.buy (b : book) = _book <- Some b
9
10  let a = author ()
11  let r = reader ()
12  let b = a.publish ()
13  r.buy (b)
```

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 **??**.
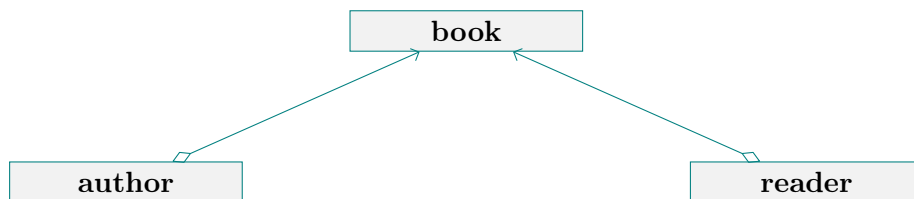
Figure 23.9: A book is an object that can be owned by both an author and a reader.

Composition
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 **??**. In this example, Owner has

depObj

| Owner | Dependent |

1..*

Figure 23.10: Composition relations are a subset of aggregation where the host controls the lifetime of the guest objects.

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 **??**. In **??**, a `dog`

Listing 23.3 umlComposition.fsx:
The `dog` object is a composition of four `leg` objects.

```
1  type leg () =
2    member this.move = "moving"
3  type dog () =
4    let _leg = List.init 4 (fun e -> leg ())
5
6  let bestFriend = 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 **??**.

| dog | leg |

Figure 23.11: A dog is a composition of legs.

23.2.2 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   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 **??**. Here two classes
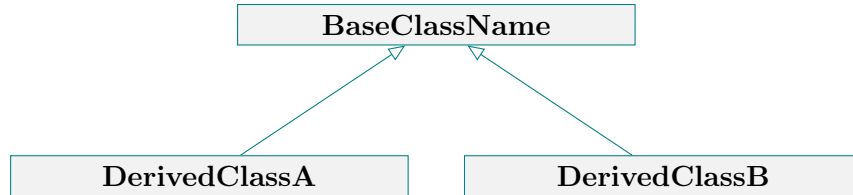
Figure 23.12: Inheritance is shown by a closed arrowhead pointing to the base.

inherit the base class.

A programming example of an inheritance is given in **??**.   In **??**, the `student` and

Listing 23.4 umlInheritance.fsx:
The student and the teacher class inherits from the person class.

```
1  type person (name : string) =
2    member this.name = name
3  type student (name : string, book : string) =
4    inherit person(name)
5    member this.book = book
6  type teacher (name : string, slides : string) =
7    inherit person(name)
8    member this.slides = slides
9
10 let s = student("Hans", "Learning to Program")
11 let t = teacher("Jon", "Slides of the day")
```

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 **??**.
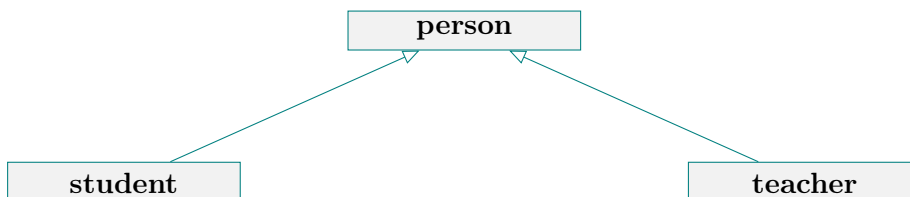
Figure 23.13: A student and a teacher inherit from a person class.

Interface 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 **??**.
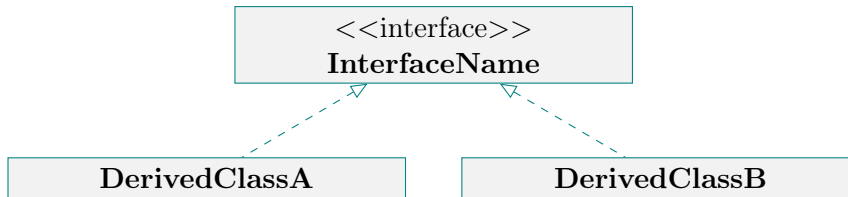
Figure 23.14: Implementations of interfaces is shown with stippled line and closed arrowhead pointing to the base.

A programming example of an interface is given in **??**. In **??**, the `television` and

Listing 23.5 umlInterface.fsx:
The `television` and the `car` class both implement the `button` interface.

```fsharp
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)
```

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 **??**.
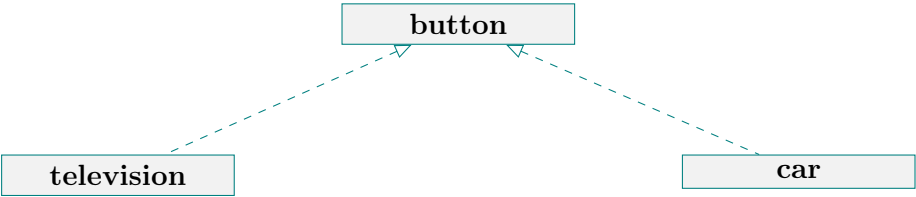
Figure 23.15: A student and a teacher inherit from a person class.

23.2.3 Packages

Namespace and modules   For visual flair, modules and namespaces are often visualized as *packages*, as shown in **??**. A package is like a module in F#.
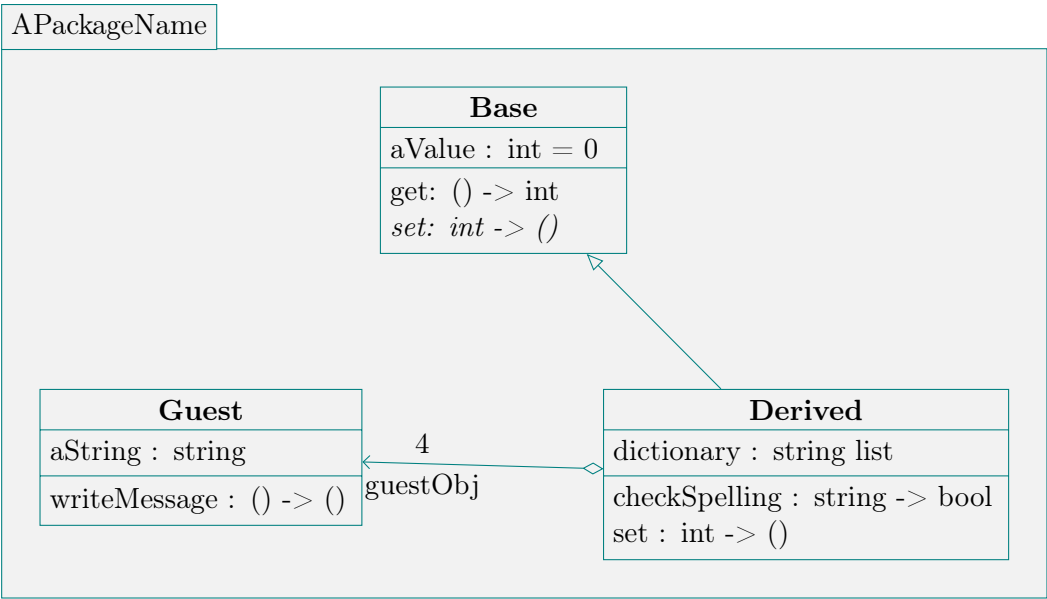
Figure 23.16: Packages are a visualizations of modules and namespaces.

23.3 Programming Intermezzo: Designing a Racing Game

An example is the following *problem statement*:

> **Task 23.1**
>
> 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.

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 ??):
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 ?? and ??):
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 ??):

A class diagram of our design for the proposed classes and their relations is shown in **??**.

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 **??**–). 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
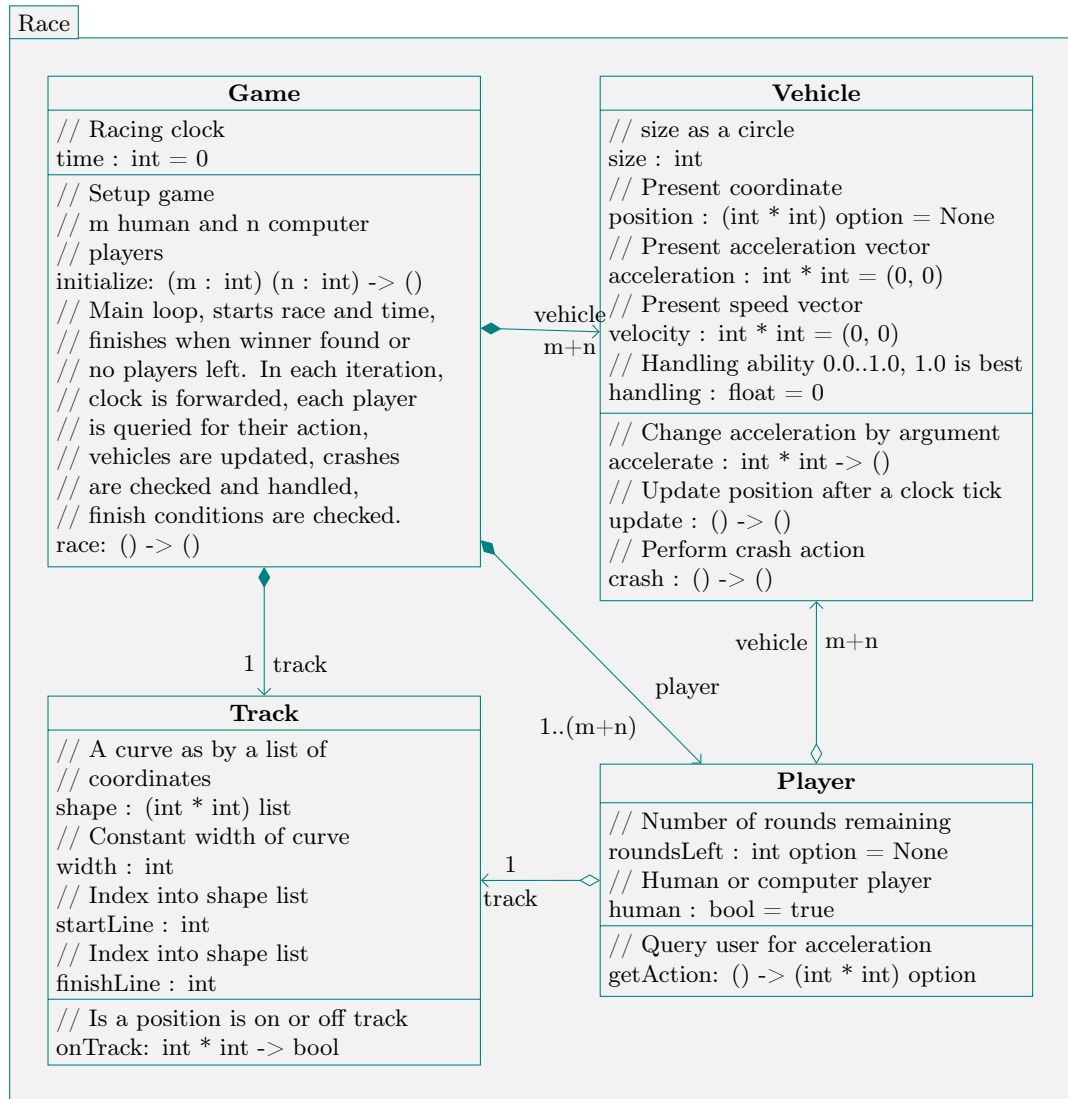
Figure 23.17: A class diagram for a racing game.

coding, since the design phase allows the programmer to think about the problem from a helicopter perspective before tackling details of specific sub-problems.

24 Graphical User Interfaces

A *graphical user interface* (*GUI*) uses graphical elements such as windows, icons, and sound to communicate with the user, and a typical way to activate these elements is through a pointing device such as the mouse or by touch. Some of these elements may themselves be textual, and thus most operating systems offer access to a *command-line interface* (*CLI*) in a window alongside other interface types.

An example of a graphical user interface is a web-browser, shown in **??**. The program
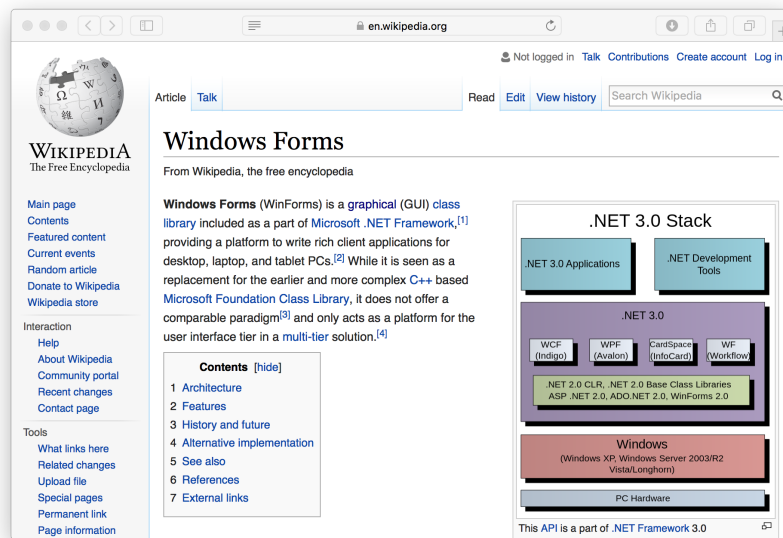
Figure 24.1: A web-browser is a graphical user interface for accessing a web-server and interacting with its services. Here the browser is showing the page `https://en.wikipedia.org/wiki/Windows_Forms` at time of writing.

presents information to the user in terms of text and images, has active areas that may be activated by clicking, allows the user to go other web-pages by typing a URL or following hyperlinks, and can generate new pages through search queries.

F# includes a number of implementations of graphical user interfaces, and at time of writing, both *GTK+* and *WinForms 2.0* are supported on both the Microsoft

.Net and the Mono platform. WinForms can be used without extra libraries during compilation, and therefore will be the subject of the following chapter.

WinForms is a set of libraries that simplifies many common tasks for applications, and in this chapter, we will focus on the graphical user interface part of WinForms. A *form* is a visual interface used to communicate information with the user, typically a window. Communication is done through *controls*, which are elements that display information or accept input. Examples of controls are a box with text, a button, and a menu. When the user gives input to a control element, this generates an *event* which you can write code to react to. WinForms is designed for *event-driven programming*, meaning that at runtime, most time is spent on waiting for the user to give input. See **??** for more on event-driven programming.

Designing easy-to-use graphical user interfaces is a challenging task. This chapter will focus on examples of basic graphical elements and how to program these in WinForms.

24.1 Opening a Window

The namespaces *System.Windows.Forms* and *System.Drawing* are central for programming graphical user interfaces with WinForms. `System.Windows.Forms` includes code for generating forms, controls, and handling events. `System.Drawing` is used for low-level drawing, and it gives access to the *Windows Graphics Device Interface* (*GDI+*), which allows you to create and manipulate graphics objects targeting several platforms, such as screens and paper. All controls in `System.Windows.Forms` in Mono are drawn using `System.Drawing`.

To display a graphical user interface on the screen, the first thing to do is open a window, which acts as a reserved screen-space for our output. In WinForms, windows are called forms. Code for opening a window is shown in **??**, and the result is shown in **??**. Note that the present version of WinForms on MacOs only works with the 32-bit implementation of mono, *mono32*, as demonstrated in the example.

   The `new` `System.Windows.Forms.Form ()` creates an object (See **??**), but does not display the window on the screen. We use the optional `new` keyword, since the form is an `IDisposable` object and may be implicitly disposed of. I.e., it is recommended to **instantiate IDisposable objects using `new` to contrast them with other object types.** Executing `System.Windows.Forms.Application.Run` is applied to the object, then the control is handed over to the WinForms' *event-loop*, which continues until the window is closed by, e.g., pressing the icon designated by the operating system. On the Mac OSX, that is the red button in the top left corner of the window frame, and on Window it is the cross on the top right corner of the window frame.

Listing 24.1 winforms/openWindow.fsx:
Create the window and turn over control to the operating system. See ??.

```
1   // Create a window
2   let win = new System.Windows.Forms.Form ()
3   // Start the event-loop.
4   System.Windows.Forms.Application.Run win
```

```
1   $ fsharpc --nologo openWindow.fsx && mono32 openWindow.exe
```
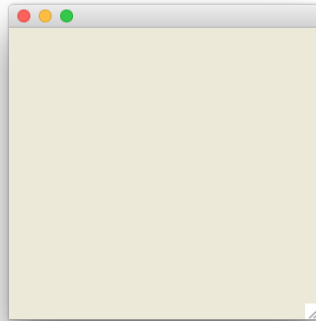
Figure 24.2: A window opened by **??**.

The window form has a long list of methods and properties. E.g., the background color may be set by `BackColor`, the title of the window may be set by `Text`, and you may get and set the size of the window with `Size`. This is demonstrated in **??**. These properties are *accessors*, implying that they act as mutable variables.

24.2 Drawing Geometric Primitives

The *System.Drawing.Color* is a structure for specifying colors as 4 channels: alpha, red, green, and blue. Some methods and properties for the Color structure is shown in **??**. Each channel is an 8-bit unsigned integer. The alpha channel specifies the transparency of a color, where values 0–255 denote the range of fully transparent to fully opaque, and the remaining channels denote the amount of red, green, and blue, where 0 is none and 255 is full intensity. As a shorthand, colors are often referred to as a single 32-bit unsigned integer, whose bits are organized in groups of 8 bits

Listing 24.2 winforms/windowProperty.fsx:
Create the window and change its properties. See ??

```
1  // Prepare window form
2  let win = new System.Windows.Forms.Form ()
3
4  // Set some properties
5  win.BackColor <- System.Drawing.Color.White
6  win.Size <- System.Drawing.Size (600, 200)
7  win.Text <- sprintf "Color '%A' and Size '%A'" win.BackColor
      win.Size
8
9  // Start the event-loop.
10 System.Windows.Forms.Application.Run win
```
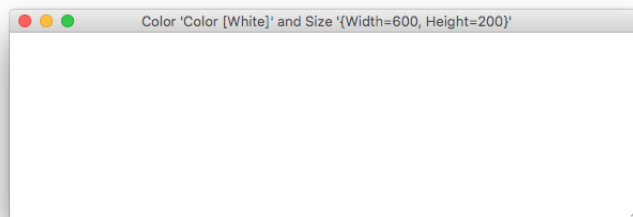

Figure 24.3: A window with user-specified size and background color, see **??**.

as 0xAARRGGBB, where AA is the alpha channel's values 0x00–0xFF etc. Any color may be created using the `FromArgb` method, e.g., an opaque red is given by `System.Drawing.Color.FromArgb (255, 255, 0, 0)`. There are also many build-in colors, e.g., the same red color is also a known color and may be obtained as `System.Drawing.Color.Red`. For a given color, the 4 alpha, red, green, and blue channels' values may be obtained as the `A`, `R`, `G`, and `B` members, see **??**

The namespace `System.Drawing` contains many useful functions and values. **??** used *System.Drawing.Size* to specify a size by a pair of integers. Other important values and functions are *Point*, which specifies a coordinate as a pair of points; *Pen*, which specifies how to draw lines and curves; *Font*, which specifies the font of a string; *SolidBrush* and *TextureBrush*, used for filling geometric primitives, and *Bitmap*, which is a type of *Image*. These are summarized in **??**.

The *System.Drawing.Graphics* is a class for drawing geometric primitives to a display device, and some of its methods are summarized in **??**.

Method/Property	Description
Properties of an existing color structure	
`A : byte`	The value of the alpha channel.
`R : byte`	The value of the red channel.
`G : byte`	The value of the green channel.
`B : byte`	The value of the blue channel.
`ToArgb : unit -> int`	The 32-bit integer value of the color.
Static properties returning a color structure by its name.	
`Black : Color`	The ARGB value `0xFF000000`.
`Blue : Color`	The ARGB value `0xFF0000FF`.
`Brown : Color`	The ARGB value `0xFFA52A2A`.
`Gray : Color`	The ARGB value `0xFF808080`.
`Green : Color`	The ARGB value `0xFF00FF00`.
`Orange : Color`	The ARGB value `0xFFFFA500`.
`Purple : Color`	The ARGB value `0xFF800080`.
`Red : Color`	The ARGB value `0xFFFF0000`.
`White : Color`	The ARGB value `0xFFFFFFFF`.
`Yellow : Color`	The ARGB value `0xFFFFFF00`.
Static methods for converting between color structures and integers representations.	
`FromArgb :` `   r:int * g:int * b:int -> Color`	Create a color structure from red, green, and blue values.

`FromArgb :`
`   a:int * r:int * g:int * b:int ->`
`Color`
	Create a color structure from alpha, red, green, and blue values.

`FromArgb : argb:int -> Color`	Create a color structure from a single integer.

Table 24.1: Some methods and properties of the `System.Drawing.Color` structure.

The location and shape of geometrical primitives are specified in a coordinate system, and WinForms operates with 2 coordinate systems: *screen coordinates* and *client coordinates*. Both coordinate systems have their origin in the top-left corner, with the first coordinate, x, increasing to the right, and the second, y, increasing down, as illustrated in **??**. The Screen coordinate system has its origin in the top-left corner of the screen, while the client coordinate system has its origin in the top-left corner of the drawable area of a form or a control, i.e., for a window, this will be the area without the window borders, scroll, and title bars. A control is a graphical object, such as a clickable button, will be discussed later. Conversion between client and screen coordinates is done with *System.Drawing.PointToClient* and *System.Drawing.PointToScreen*.

Listing 24.3 drawingColors.fsx:
Defining colors and accessing their values.

```
1   // open namespace for brevity
2   open System.Drawing
3   // Define a color from ARGB
4   let c = Color.FromArgb (0xFF, 0x7F, 0xFF, 0xD4) //Aquamarine
5   printfn "The color %A is (%x, %x, %x, %x)" c c.A c.R c.G c.B
6   // Define a list of named colors
7   let colors =
8     [Color.Red; Color.Green; Color.Blue;
9      Color.Black; Color.Gray; Color.White]
10  for col in colors do
11    printfn "The color %A is (%x, %x, %x, %x)" col col.A col.R
        col.G col.B
```

```
1   $ fsharpc --nologo drawingColors.fsx && mono
       drawingColors.exe
2   The color Color [A=255, R=127, G=255, B=212] is (ff, 7f,
       ff, d4)
3   The color Color [Red] is (ff, ff, 0, 0)
4   The color Color [Green] is (ff, 0, 80, 0)
5   The color Color [Blue] is (ff, 0, 0, ff)
6   The color Color [Black] is (ff, 0, 0, 0)
7   The color Color [Gray] is (ff, 80, 80, 80)
8   The color Color [White] is (ff, ff, ff, ff)
```
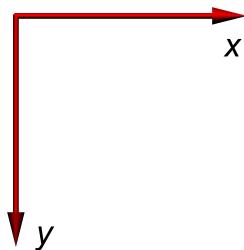

Figure 24.4: Coordinate systems in Winforms have the y axis pointing down.

Displaying graphics in WinForms is performed as the reaction to an event. E.g., windows are created by the program, moved, minimized, occluded by other windows, resized, etc., by the user or the program, and each action may require that the content of the window is refreshed. Thus, we must create a function that WinForms can call any time. This is known as a *call-back function*, and it is added to an existing form using the form's **Paint.Add** method. Due to the event-driven nature of WinForms,

Constructor	Description
`Bitmap(int, int)`	Create a new empty `Image` of specified size.
`Bitmap(Stream)` `Bitmap(string)`	Create a `Image` from a `System.IO.Stream` or from a file specified by a filename.
`Font(string, single)`	Create a new font from the font's name and em-size.

`Pen(Brush)`
`Pen(Brush), single)`
`Pen(Color)`
`Pen(Color, single)`
	Create a pen to paint either with a brush or solid color and possibly with specified width.

`Point(int, int)`
`Point(Size)`
`PointF(single, single)`
	Create an ordered pair of integers or singles specifying x- and y-coordinates in the plane.

`Size(int, int)`
`Size(Point)`
`SizeF(single, single)`
`SizeF(PointF)`
	Create an ordered pair of integers or singles specifying height and width in the plane.

`SolidBrush(Color)` `TextureBrush(Image)`	Create a `Brush` as a solid color or from an image to fill the interior of geometric shapes.

Table 24.2: Basic geometrical structures in WinForms. `Brush` and `Image` are abstract classes.

functions for drawing graphics primitives are only available when responding to an event, e.g., *System.Drawing.Graphics.DrawLines* draws a line in a window, and *it is only possible to call this function as part of an event handling.*

As an example, consider the problem of drawing a triangle in a window. For this we need to make a function that can draw a triangle not once, but at any amount of times as deemed necessary by the operating system. An example of such a program is shown in **??**. A walk-through of the code is as follows: First, we open the two libraries that we will use heavily. This will save us some typing, but also pollute our namespace. E.g., now `Point` and `Color` are existing types, and we cannot define our own identifiers with these names. Then we create the form with size 320 × 170, we add a paint call-back function, and we start the event-loop. The event-loop will call the paint function, whenever the system determines that the window's content needs to be refreshed. This function is to be called as a response to a paint event and takes a `System.Windows.Forms.PaintEventArgs` object, which includes the System.Drawing.Graphics object. The function `paint` chooses a pen and a set of

Constructor	Description
`DrawImage : Image * (Point []) -> unit` `DrawImage : Image * (PointF []) -> unit`	Draw an image at a specific point and size.
`DrawImage : Image * Point -> unit` `DrawImage : Image * PointF -> unit`	Draw an image at a specific point.
`DrawLines : Pen * (Point []) -> unit` `DrawLines : Pen * (PointF []) -> unit`	Draw a series of lines between the n'th and $n+1$'th points.

`DrawString :`
`   string * Font * Brush * PointF ->`
`   unit`
	Draw a string at the specified point.

Table 24.3: Basic geometrical structures in WinForms.
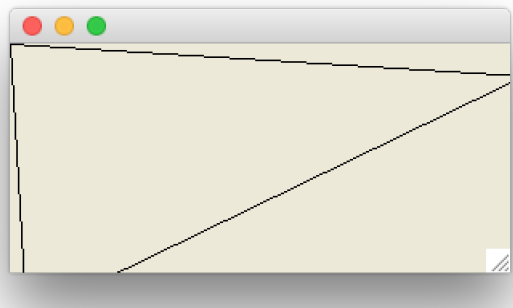
Figure 24.5: Drawing a triangle using **??**.

points and draws a set of lines connecting the points.

The code in **??** is not optimal. Despite the fact that the triangle spans the rectangle $(0,0)$ to $(320, 170)$ and the window's size is set to $(320, 170)$, our window is too small and the triangle is clipped at the window border. The error is that we set the window's `Size` property, which determines the size of the window including top bar and borders. Alternatively, we may set the `ClientSize`, which determines the size of the drawable area, and this is demonstrated in **??** and **??**.   Thus, **prefer the**   ⋆
ClientSize over the `Size` **property for internal consistency.**

Considering the program in **??**, we may identify a part that concerns the specification of the triangle, or more generally the graphical model, and some which concern system

Listing 24.4 winforms/triangle.fsx:
Adding line graphics to a window. See ??

```
1   // Open often used libraries , beware of namespace polution!
2   open System.Windows.Forms
3   open System.Drawing
4
5   // Prepare window form
6   let win = new Form ()
7   win.Size <- Size (320, 170)
8
9   // Set paint call -back function
10  let paint (e : PaintEventArgs) : unit =
11    let pen = new Pen (Color.Black)
12    let points =
13      [|Point (0,0); Point (10,170); Point (320,20); Point (0,0)|]
14    e.Graphics.DrawLines (pen, points)
15  win.Paint.Add paint
16
17  // Start the event -loop.
18  Application.Run win
```
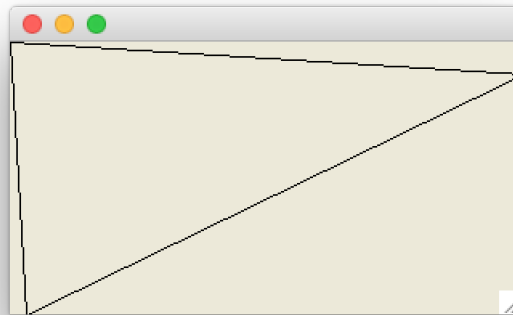

Figure 24.6: Setting the ClientSize property gives a predictable drawing area, see **??** for code.

specific details. For future maintenance, it is often a good idea to **separate the** ⋆ **model from how it is viewed on a specific system**. E.g., it may be that at some point you decide that you would rather use a different library than WinForms. In this case, the general graphical model will be the same, but the specific details on initialization and event handling will be different. We think of the model and the viewing part of the code as top and bottom layers, respectively, and these are often

> **Listing 24.5 winforms/triangleClientSize.fsx:**
> **Adding line graphics to a window. See ??.**

```fsharp
1   // Open often used libraries, beware of namespace polution!
2   open System.Windows.Forms
3   open System.Drawing
4
5   // Prepare window form
6   let win = new Form ()
7   win.ClientSize <- Size (320, 170)
8
9   // Set paint call-back function
10  let paint (e : PaintEventArgs) : unit =
11    let pen = new Pen (Color.Black)
12    let points =
13      [|Point (0,0); Point (10,170); Point (320,20); Point (0,0)|]
14    e.Graphics.DrawLines (pen, points)
15  win.Paint.Add paint
16
17  // Start the event-loop.
18  Application.Run win
```

connected with a connection layer. This *Model-View paradigm* is shown in **??**. While
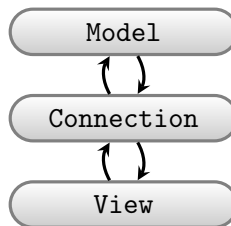
Figure 24.7: Separating model from view gives flexibility later.

it is not easy to completely separate the general from the specific, it is often a good idea to strive for some degree of separation.

In **??**, the program has been redesigned to follow the Model-View paradigm, where `view` contains most of the WinForms-specific code, and `model` contains most of the geometry, which could be reused with other graphical user interfaces. The model still uses the geometric primitives from WinForms for brevity, since a general implementation of geometric primitives avoiding WinForms would have a very similar interface. This program is longer, but there is a much better separation of *what* is to be displayed (model) from *how* it is to be done (view).

To further our development of a general program for displaying graphics, consider the

Listing 24.6 winforms/triangleOrganized.fsx:
Improved organization of code for drawing a triangle. See ??.

```
1   // Open often used libraries, beware of namespace polution!
2   open System.Windows.Forms
3   open System.Drawing
4
5   ///////////// WinForm specifics /////////////
6   /// Setup a window form and return function which can activate it
7   let view (sz : Size) (pen : Pen) (pts : Point []) : (unit ->
       unit) =
8     let win = new System.Windows.Forms.Form ()
9     win.ClientSize <- sz
10    win.Paint.Add (fun e -> e.Graphics.DrawLines (pen, pts))
11    fun () -> Application.Run win // function as return value
12
13  ///////////// Model /////////////
14  // A black triangle, using winform primitives for brevity
15  let model () : Size * Pen * (Point []) =
16    let size = Size (320, 170)
17    let pen = new Pen (Color.FromArgb (0, 0, 0))
18    let lines =
19      [|Point (0,0); Point (10,170); Point (320,20); Point (0,0)|]
20    (size, pen, lines)
21
22  ///////////// Connection /////////////
23  // Tie view and model together and enter main event loop
24  let (size, pen, lines) = model ()
25  let run = view size pen lines
26  run ()
```

case where we are to draw another two triangles, that are a translation and rotations of the original, and where we would like to specify the color of each triangle individually. A simple extension of `model` in **??** for generating many shapes of different colors is `model : unit -> Size * ((Point []) * Pen) list`, i.e., semantically augment each point array with a pen and return a list of such pairs. For this example, we also program translation and rotation transformations. See **??** for the result. We update `view` accordingly to iterate through this list as shown in **??**. Since we are using WinForms primitives in the model, the connection layer is trivial, as shown in **??**.
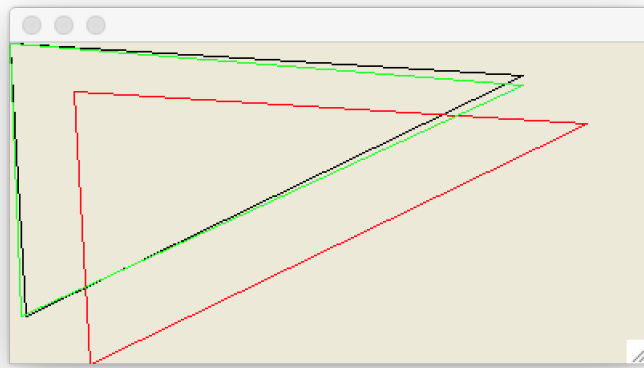
Figure 24.8: Better organization of the code for drawing a triangle, see **??**.

Listing 24.7 winforms/transformWindows.fsx:
Model of a triangle and simple transformations of it. See also ????.

```
15  ////////////// Model //////////////
16  // A black triangle, using WinForm primitives for brevity
17  let model () : Size * ((Pen * (Point [])) list) =
18    /// Translate a primitive
19    let translate (d : Point) (arr : Point []) : Point [] =
20      let add (d : Point) (p : Point) : Point =
21        Point (d.X + p.X, d.Y + p.Y)
22      Array.map (add d) arr
23
24    /// Rotate a primitive
25    let rotate (theta : float) (arr : Point []) : Point [] =
26      let toInt = int << round
27      let rot (t : float) (p : Point) : Point =
28        let (x, y) = (float p.X, float p.Y)
29        let (a, b) = (x * cos t - y * sin t, x * sin t + y * cos t)
30        Point (toInt a, toInt b)
31      Array.map (rot theta) arr
32
33    let size = Size (400, 200)
34    let lines =
35      [|Point (0,0); Point (10,170); Point (320,20); Point (0,0)|]
36    let black = new Pen (Color.FromArgb (0, 0, 0))
37    let red = new Pen (Color.FromArgb (255, 0, 0))
38    let green = new Pen (Color.FromArgb (0, 255, 0))
39    let shapes =
40      [(black, lines);
41       (red, translate (Point (40, 30)) lines);
42       (green, rotate (1.0 *System.Math.PI / 180.0) lines)]
43    (size, shapes)
```

Listing 24.8 winforms/transformWindows.fsx:
A view for lists of pairs of pen and point arrays. See also ????.

```fsharp
// Open often used libraries, beware of namespace polution!
open System.Windows.Forms
open System.Drawing

///////////// WinForm specifics /////////////
/// Setup a window form and return function to activate
let view (sz : Size) (shapes : (Pen * (Point [])) list) : (unit
   -> unit) =
  let win = new Form ()
  win.ClientSize <- sz
  let paint (e : PaintEventArgs) ((p, pts) : (Pen * (Point [])))
   : unit =
    e.Graphics.DrawLines (p, pts)
  win.Paint.Add (fun e -> List.iter (paint e) shapes)
  fun () -> Application.Run win // function as return value
```

Listing 24.9 winforms/transformWindows.fsx:
Model of a triangle and simple transformations of it. See also ????.

```fsharp
///////////// Connection /////////////
// Tie view and model together and enter main event loop
let (size, shapes) = model ()
let run = view size shapes
run ()
```

24.3 Programming Intermezzo: Hilbert Curve

A curve in 2 dimensions has a length but no width, and we can only visualize it by giving it a width. Thus, it came as a surprise to many when Giuseppe Peano in 1890 demonstrated that there exist curves which fill every point in a square. The method he used to achieve this was recursion:

Task 24.1

Consider a curve consisting of piecewise straight lines all with the same length but with varying angles 0°, 90°, 180°, or 270° w.r.t. the horizontal axis. To draw this curve, we need 3 basic operations: Move forward (F), turn right (R), and turn left (L). The turning is w.r.t. the present direction. A Hilbert Curve is a space-filling curve which can be expressed recursively as:

$$A \to LBFRAFARFBL, \qquad (24.17)$$
$$B \to RAFLBFBLFAR, \qquad (24.18)$$

starting with A. For practical illustrations, we typically only draw space-filling curves to a specified depth of recursion, which is called the order of the curve. To keep track of the level of recursion, we introduce an index as:

$$A_{n+1} \to LB_nFRA_nFA_nRFB_nL,$$
$$B_{n+1} \to RA_nFLB_nFB_nLFA_nR,$$

for $n > 0$ and $A_0 \to \emptyset$ and $B_0 \to \emptyset$. Thus, the first-order curve is

$$A_1 \to LB_0FRA_0FA_0RFB_0L \to LFRFRFL,$$

and the second order curve is

$$A_2 \to LB_1FRA_1FA_1RFB_1L$$
$$\to LRFLFLFRFRLFRFRFLFLFRFRFLRFRFLFLFRL.$$

Since $LR = RL = \emptyset$ the above simplifies to

$$A_2 \to FLFLFRFFRFRFLFLFRFRFFRFLFLF$$

Make a program that given an order produces an image of the Hilbert curve.

Our strategy to solve this problem will be first to define the curves in terms of movement commands $LRFL\ldots$. For this, we will define a discriminated union `type Command = F | L | R`. The movement commands can then be defined as a `Command list` type. The list for a specific order is a simple set of recursive functions in F#

which we will call `A` and `B`.

To produce a graphical drawing of a command list, we must transform it into coordinates, and during the conversion, we need to keep track of both the present position and the present heading, since not all commands draw. This is a concept similar to Turtle Graphics, which is often associated with the Logo programming language from the 1960's. In Turtle graphics, we command a little robot - a turtle - which moves in 2 dimensions and can turn on the spot or move forward, and its track is the line being drawn. Thus we introduce a `type Turtle = {x : float; y : float; d : float}` record. Conversion of command lists to turtle lists is a fold programming structure, where the command list is read from left-to-right, building up an accumulator by adding each new element. For efficiency, we choose to prepend the new element to the accumulator. This we have implemented as the `addRev` function. Once the full list of turtles has been produced, then it is reversed.

Finally, the turtle list is converted to WinForms `Point` array, and a window of appropriate size is chosen. The resulting model part is shown in **??**. The view and connection parts are identical to **????**, and **??** shows the result of using the program to draw Hilbert curves of orders 1, 2, 3, and 5.

(a) Order 1   (b) Order 2   (c) Order 3

(d) Order 5
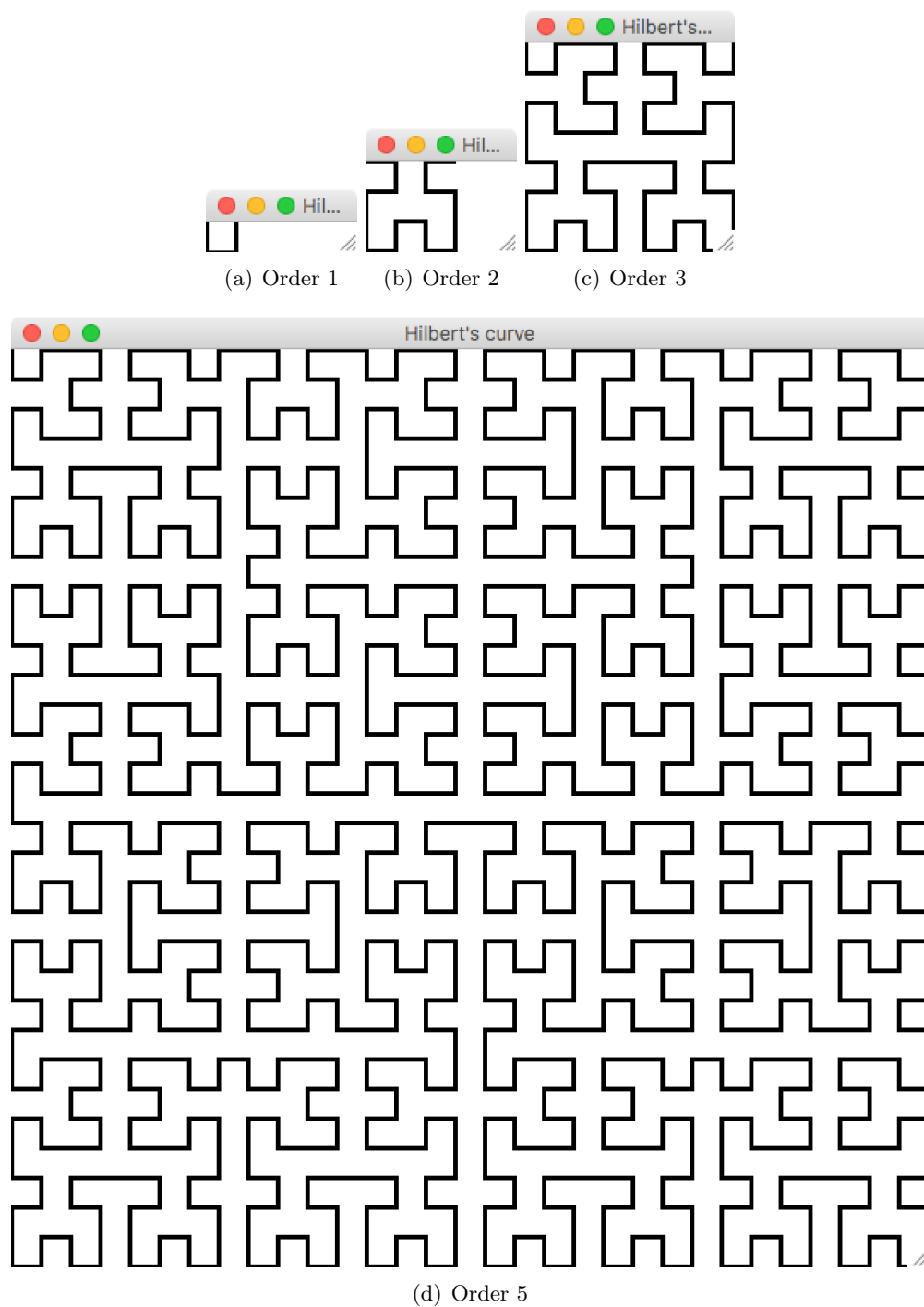
Figure 24.9: Hilbert curves of orders 1, 2, 3, and 5 by code in **??**.

> **Listing 24.10 winforms/hilbert.fsx:**
> Using simple turtle graphics to produce a list of points on a polygon. The code continues in Listing ??. The view and connection parts are identical to ????.

```fsharp
15  // Turtle commands, type definitions must be in outermost scope
16  type Command = F | L | R
17  type Turtle = {x : float; y : float; d : float}
18
19  // A black Hilbert curve using WinForm primitives for brevity
20  let model () : Size * ((Pen * (Point [])) list) =
21    /// Hilbert recursion production rules
22    let rec A n : Command list =
23      if n > 0 then
24        [L]@B (n-1)@[F; R]@A (n-1)@[F]@A (n-1)@[R; F]@B (n-1)@[L]
25      else
26        []
27    and B n : Command list =
28      if n > 0 then
29        [R]@A (n-1)@[F; L]@B (n-1)@[F]@B (n-1)@[L; F]@A (n-1)@[R]
30      else
31        []
32
33    /// Convert a command to turtle record and prepend to list
34    let addRev (lst : Turtle list) (cmd : Command) (len : float) :
       Turtle list =
35      let toInt = int << round
36      match lst with
37        | t::rest ->
38          match cmd with
39            | L -> {t with d = t.d + 3.141592/2.0}::rest // left
40            | R -> {t with d = t.d - 3.141592/2.0}::rest // right
41            | F -> {t with x = t.x + len * cos t.d; // forward
42                           y = t.y + len * sin t.d}::lst
43        | _ -> failwith "Turtle list must be non-empty."
44
45    let maxPoint (p1 : Point) (p2 : Point) : Point =
46      Point (max p1.X p2.X, max p1.Y p2.Y)
```

Listing 24.11 winforms/hilbert.fsx:
Continued from Listing ??.

```
48    // Calculate commands for a specific order
49    let curve = A 5
50    // Convert commands to point array
51    let initTrtl = {x = 0.0; y = 0.0; d = 0.0}
52    let len = 20.0
53    let line =
54      List.fold (fun acc elm -> addRev acc elm len) [initTrtl]
     curve // Convert command list to reverse turtle list
55      |> List.rev // Reverse list
56      |> List.map (fun t -> Point (int (round t.x), int (round
     t.y))) // Convert turtle list to point list
57      |> List.toArray // Convert point list to point array
58    let black = new Pen (Color.FromArgb (0, 0, 0))
59    // Set size to as large as shape
60    let minVal = System.Int32.MinValue
61    let maxPoint = Array.fold maxPoint (Point (minVal, minVal)) line
62    let size = Size (maxPoint.X + 1, maxPoint.Y + 1)
63    (size, [(black, line)]) // return shapes as singleton list
```

24.4 Handling Events

In the previous section, we have looked at how to draw graphics using the `Paint` method of an existing form object. Forms have many other event handlers that we may use to interact with the user. **??** demonstrates event handlers for moving and resizing a window, for clicking in a window, and for typing on the keyboard. **??** shows the output from an interaction with the program which is the result of the following actions: moving the window, resizing the window, clicking the left mouse key, pressing and holding the down the left mouse key while moving the mouse, releasing the left mouse key, and typing "abc". As demonstrated, some actions, like moving the mouse, result in a lot of events, and some, like the initial window moves results, are surprising. Thus, event-driven programming should take care to interpret the events robustly and carefully.

Common for all event-handlers is that they listen for an event, and when the event occurs, the functions that have been added using the `Add` method are called. This is also known as sending a message. Thus, a single event can give rise to calling zero or more functions.

Graphical user interfaces and other systems often need to perform actions that depend on specific lengths of time or a certain point in time. To measure length of time F# has the *System.Windows.Forms.Timer* class, which technically is an optimized of `System.Timers.Timer` for graphical user interfaces. The Timer class can be used to create an event after a specified duration of time. F# also has the *System.DateTime* class to specify points in time. An often used property is `System.DateTime.Now`, which returns a `DateTime` object for the date and time when the property is accessed. The use of these two classes is demonstrated in **??** and **??**. In the code, a label has
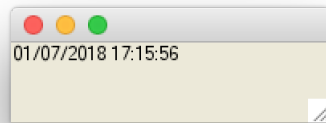
Figure 24.10: See **??**.

been created to show the present date and time. The label is a type of control, and it is displayed using the default font which is rather small. How to change this and other details on controls will be discussed in the next section.

In the example, the label is redrawn everytime the text is changed, such that the current value is correctly displayed on the screen. Sometimes it is necessary to force a control to redraw which can be done with the `Refresh()` method. Since a `Form` is also a type of control, it is common to trigger a redraw event for the top form, which in **??** would be `win.Refresh()`. Thus, `Refresh()` and a `Timer` object can be used to produce animations.

Listing 24.12 winforms/windowEvents.fsx:
Catching window, mouse, and keyboard events.

```
1   open System.Windows.Forms
2   open System.Drawing
3   open System
4
5   let win = new Form () // create a form
6
7   // Window event
8   let windowMove (e : EventArgs) =
9     printfn "Move: %A" win.Location
10  win.Move.Add windowMove
11
12  let windowResize (e : EventArgs) =
13    printfn "Resize: %A" win.DisplayRectangle
14  win.Resize.Add windowResize
15
16  // Mouse event
17  let mutable record = false; // records when button down
18  let mouseMove (e : MouseEventArgs) =
19    if record then printfn "MouseMove: %A" e.Location
20  win.MouseMove.Add mouseMove
21
22  let mouseDown (e : MouseEventArgs) =
23    printfn "MouseDown: %A" e.Location; (record <- true)
24  win.MouseDown.Add mouseDown
25
26  let mouseUp (e : MouseEventArgs) =
27    printfn "MouseUp: %A" e.Location; (record <- false)
28  win.MouseUp.Add mouseUp
29
30  let mouseClick (e : MouseEventArgs) =
31    printfn "MouseClick: %A" e.Location
32  win.MouseClick.Add mouseClick
33
34  // Keyboard event
35  win.KeyPreview <- true
36  let keyPress (e : KeyPressEventArgs) =
37    printfn "KeyPress: %A" (e.KeyChar.ToString ())
38  win.KeyPress.Add keyPress
39
40  Application.Run win // Start the event-loop.
```

Listing 24.13: Output from an interaction with the program in ??.

```
1   Move: {X=22,Y=22}
2   Move: {X=22,Y=22}
3   Move: {X=50,Y=71}
4   Resize: {X=0,Y=0,Width=307,Height=290}
5   MouseDown: {X=144,Y=118}
6   MouseClick: {X=144,Y=118}
7   MouseUp: {X=144,Y=118}
8   MouseDown: {X=144,Y=118}
9   MouseUp: {X=144,Y=118}
10  MouseDown: {X=96,Y=66}
11  MouseMove: {X=96,Y=67}
12  MouseMove: {X=97,Y=69}
13  MouseMove: {X=99,Y=71}
14  MouseMove: {X=103,Y=74}
15  MouseMove: {X=107,Y=77}
16  MouseMove: {X=109,Y=79}
17  MouseMove: {X=112,Y=81}
18  MouseMove: {X=114,Y=82}
19  MouseMove: {X=116,Y=84}
20  MouseMove: {X=117,Y=85}
21  MouseMove: {X=118,Y=85}
22  MouseClick: {X=118,Y=85}
23  MouseUp: {X=118,Y=85}
24  KeyPress: "a"
25  KeyPress: "b"
26  KeyPress: "c"
```

Listing 24.14 winforms/clock.fsx:
Using `System.Windows.Forms.Timer` and `System.DateTime.Now` to update the display of the present date and time. See ?? for the result.

```
1   open System.Windows.Forms
2   open System.Drawing
3   open System
4
5   let win = new Form () // make a window form
6   win.ClientSize <- Size (200, 50)
7
8   // make a label to show time
9   let label = new Label()
10  win.Controls.Add label
11  label.Width <- 200
12  label.Text <- string System.DateTime.Now // get present time and
        date
13
14  // make a timer and link to label
15  let timer = new Timer ()
16  timer.Interval <- 1000 // create an event every 1000 millisecond
17  timer.Enabled <- true // activate the timer
18  timer.Tick.Add (fun e -> label.Text <- string System.DateTime.Now)
19
20  Application.Run win // start event-loop
```

24.5 Labels, Buttons, and Pop-up Windows

In WinForms, buttons, menus and other interactive elements are called *Controls*. A form is a type of control, and thus, programming controls are very similar to programming windows. **??** shows a small program that displays a label and a button in a window, and when the button is pressed, then the label is updated.   As the listing

> **Listing 24.15 winforms/buttonControl.fsx:**
> **Create the button and an event, see also ??.**
>
> ```
>  1  open System.Windows.Forms
>  2  open System.Drawing
>  3  open System
>  4
>  5  let win = new Form () // make a window form
>  6  win.ClientSize <- Size (140, 120)
>  7
>  8  // Create a label
>  9  let label = new Label()
> 10  win.Controls.Add label
> 11  label.Location <- new Point (20, 20)
> 12  label.Width <- 120
> 13  let mutable clicked = 0
> 14  let setLabel clicked =
> 15    label.Text <- sprintf "Clicked %d times" clicked
> 16  setLabel clicked
> 17
> 18  // Create a button
> 19  let button = new Button ()
> 20  win.Controls.Add button
> 21  button.Size <- new Size (100, 40)
> 22  button.Location <- new Point (20, 60)
> 23  button.Text <- "Click me"
> 24  button.Click.Add (fun e -> clicked <- clicked + 1; setLabel
>       clicked)
> 25
> 26  Application.Run win // Start the event-loop.
> ```

demonstrates, the button is created using the `System.Windows.Forms.Button` constructor, and it is added to the window's form's control list. The `Location` property controls its position w.r.t. the enclosing form. Other accessors are `Width`, `Text`, and `Size`.

`System.Windows.Forms` includes a long list of controls, some of which are summarized in **??**. Examples are given in `controls`, shown in **??** and **??**.

Some controls open separate windows for more involved dialogue with the user. Some
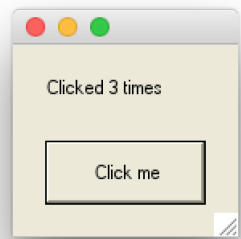
Figure 24.11: After pressing the button 3 times. See **??**.

Method/Property	Description
`Button`	A clickable button.
`CheckBox`	A clickable check box.
`DateTimePicker`	A box showing a date with a drop-down menu for choosing another.
`Label`	A displayable text.
`ProgressBar`	A box showing a progress bar.
`RadioButton`	A single clickable radio button. Can be paired with other radio buttons.
`TextBox`	A text area, which can accept input from the user.

Table 24.4: Some types of `System.Windows.Forms.Control`.

examples are `MessageBox`, `OpenFileDialog`, and `SaveFileDialog`.

System.Windows.Forms.MessageBox is used to have a simple but restrictive dialogue with the user, which is demonstrated in **??** and **??**. As an alternative to the `YesNo` response button, the message box also offers `AbortRetryIgnore`, `OK`, `OKCancel`, `RetryCancel`, and `YesNoCancel`. Note that all other windows of the process are blocked until the user closes the dialogue window.

With *System.Windows.Forms.OpenFileDialog*, you can ask the user to select an existing filename, as demonstrated in **??** and **??**. Similarly to `OpenFileDialog`, *System.Windows.Forms.SaveFileDialog* asks for a file name, but if an existing file is selected, then the user will be asked to confirm the choice.
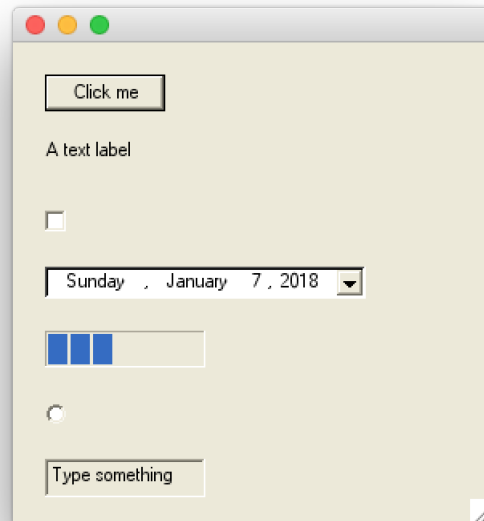
Figure 24.12: Examples of control elements. See **??**.
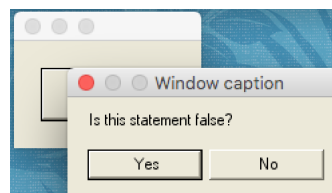
Figure 24.13: After pressing the "Click-me" button. See **??**.

Listing 24.16 winforms/controls.fsx:
Examples of control elements added to a window form, see also ??.

```fsharp
open System.Windows.Forms
open System.Drawing

let win = new Form () // Create a window
win.ClientSize <- Size (300, 300)

let button = new Button () // Make a button
win.Controls.Add button
button.Location <- new Point (20, 20)
button.Text <- "Click me"

let lbl = new Label () // Add a label
win.Controls.Add lbl
lbl.Location <- new Point (20, 60)
lbl.Text <- "A text label"

let chkbox = new CheckBox () // Add a check box
win.Controls.Add chkbox
chkbox.Location <- new Point (20, 100)

let pick = new DateTimePicker () // Add a date and time picker
win.Controls.Add pick
pick.Location <- new Point (20, 140)

let prgrss = new ProgressBar () // Show a progress bar
win.Controls.Add prgrss
prgrss.Location <- new Point (20, 180)
prgrss.Minimum <- 0
prgrss.Maximum <- 9
prgrss.Value <- 3

let rdbttn = new RadioButton () // Add a radio button
win.Controls.Add rdbttn
rdbttn.Location <- new Point (20, 220)

let txtbox = new TextBox () // Add a text input field
win.Controls.Add txtbox
txtbox.Location <- new Point (20, 260)
txtbox.Text <- "Type something"

Application.Run win // Show everything and start event-loop
```

Listing 24.17 winforms/messageBox.fsx:
Create the `MessageBox`, see also **??**.

```
1  open System.Windows.Forms
2  open System.Drawing
3  open System
4
5  let win = new Form ()
6  win.ClientSize <- Size (140, 80)
7
8  let button = new Button ()
9  win.Controls.Add button
10  button.Size <- new Size (100, 40)
11  button.Location <- new Point (20, 20)
12  button.Text <- "Click me"
13  // Open a message box when button clicked
14  let buttonClicked (e : EventArgs) =
15    let question = "Is this statement false?"
16    let caption = "Window caption"
17    let boxType = MessageBoxButtons.YesNo
18    let response = MessageBox.Show (question, caption, boxType)
19    printfn "The user pressed %A" response
20  button.Click.Add buttonClicked
21
22  Application.Run win
```
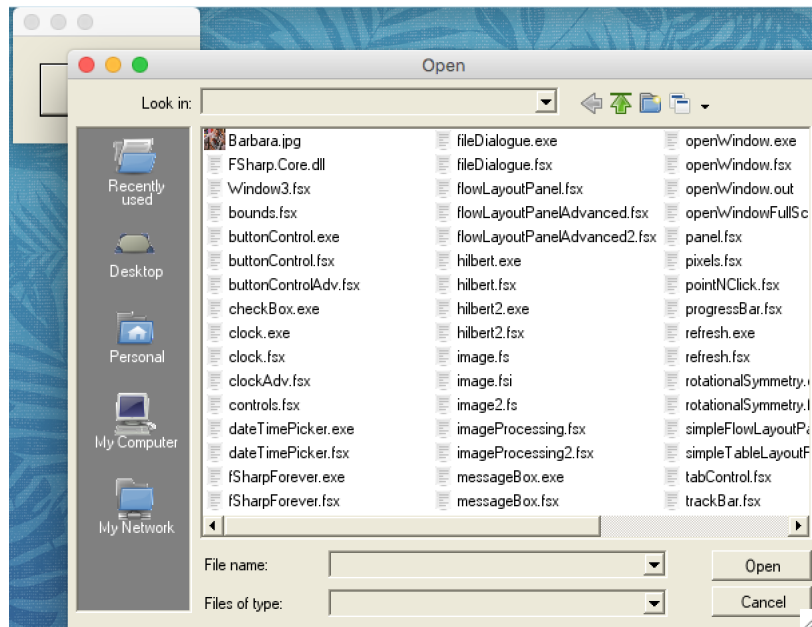

Figure 24.14: Ask the user for a filename to read from. See **??**.

Listing 24.18 winforms/openFileDialog.fsx:
Create the `OpenFileDialog`, see also ??.

```
open System.Windows.Forms
open System.Drawing
open System

let win = new Form ()
win.ClientSize <- Size (140, 80)

let button = new Button ()
win.Controls.Add button
button.Size <- new Size (100, 40)
button.Location <- new Point (20, 20)
button.Text <- "Click me"
// Open a message box when button clicked
let buttonClicked (e : EventArgs) =
  let opendlg = new OpenFileDialog()
  let okOrCancel = opendlg.ShowDialog()
  printfn "The user pressed %A and selected %A" okOrCancel
   opendlg.FileName
button.Click.Add buttonClicked

Application.Run win
```

24.6 Organizing Controls

It is often useful to organize the controls in groups, and such groups are called *Panels* in WinForms. An example of creating a `System.Windows.Forms.Panel` that includes a `System.Windows.Forms.TextBox` and `System.Windows.Forms.Label` for getting user input is shown in **??** and **??**. The label and textbox are children of the panel,

---

Listing 24.19 winforms/panel.fsx:
Create a panel, label, and text input controls.

```
1   open System.Drawing
2   open System.Windows.Forms
3
4   let win = new Form () // Create a window form
5   win.ClientSize <- new Size (200, 100)
6
7   // Customize the Panel control
8   let panel = new Panel ()
9   panel.ClientSize <- new Size (160, 60)
10  panel.Location <- new Point (20,20)
11  panel.BorderStyle <- BorderStyle.Fixed3D
12  win.Controls.Add panel // Add panel to window
13
14  // Customize the Label and TextBox controls
15  let label = new Label ()
16  label.ClientSize <- new Size (120, 20)
17  label.Location <- new Point (15,5)
18  label.Text <- "Input"
19  panel.Controls.Add label // add label to panel
20
21  let textBox = new TextBox ()
22  textBox.ClientSize <- new Size (120, 20)
23  textBox.Location <- new Point (20,25)
24  textBox.Text <- "Initial text"
25  panel.Controls.Add textBox // add textbox to panel
26
27  Application.Run win // Start the event-loop
```

---

and the main advantage of using panels is that the coordinates of the children are relative to the top left corner of the panel. I.e., moving the panel will move the label and the textbox at the same time.

A very flexible panel is the *System.Windows.Forms.FlowLayoutPanel*, which arranges its objects according to the space available. This is useful for graphical user interfaces targeting varying device sizes, such as a computer monitor and a tablet, and it also allows the program to gracefully adapt when the user changes window size. A demonstration of `System.Windows.Forms.FlowLayoutPanel` together
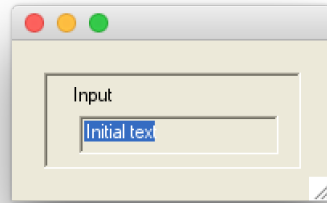
Figure 24.15: A panel including a label and a text input field, see **??**.

with `System.Windows.Forms.CheckBox` and `System.Windows.Forms.RadioButton` is given in **??**–**??** and in **??**. The program illustrates how the button elements flow under four possible flow directions with `System.Windows.FlowDirection`, and how `System.Windows.WrapContents` influences what happens to content that flows outside the panel's region. A walkthrough of the program is as follows. The goal is
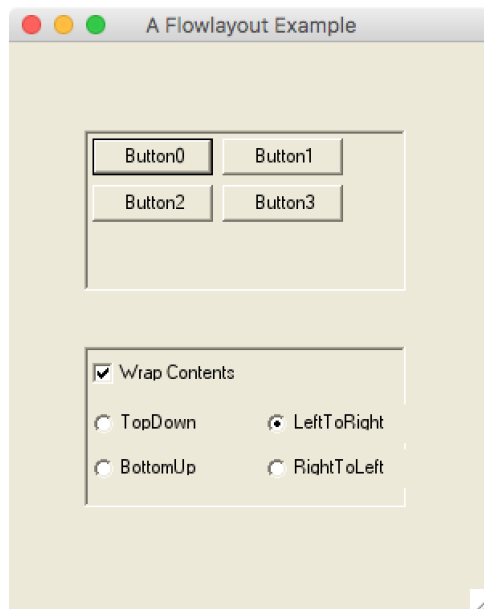
Figure 24.16: Demonstration of the `FlowLayoutPanel` panel, `CheckBox`, and `RadioButton` controls, see **??**–**??**.

to make 2 areas, one giving the user control over display parameters, and another displaying the result of the user's choices. These are `FlowLayoutPanel` and `Panel`. In the `FloatLayoutPanel` there are four `Button`s to be displayed in a region that

is not tall enough for the buttons to be shown in vertical sequence and not wide
enough to be shown in horizontal sequence. Thus the `FlowDirection` rules come
into play, i.e., the buttons are added in sequence as they are named, and the default
`FlowDirection.LeftToRight` arranges the `buttonLst.[0]` in the top left corner, and
`buttonLst.[1]` to its right. Other flow directions do it differently, and the reader is
encouraged to experiment with the program.

The program in **??** has not completely separated the semantic blocks of the interface
and relies on explicit setting of coordinates of controls. This can be avoided by using
nested panels. E.g., in **??**–**??**, the program has been rewritten as a nested set of
`FloatLayoutPanel` in three groups: The button panel, the checkbox, and the radio
button panel. Adding a `Resize` event handler for the window to resize the outermost
panel according to the outer window allows for the three groups to change position
relative to each other. This results in three different views, all shown in **??**.

(a)   (b)

(c)
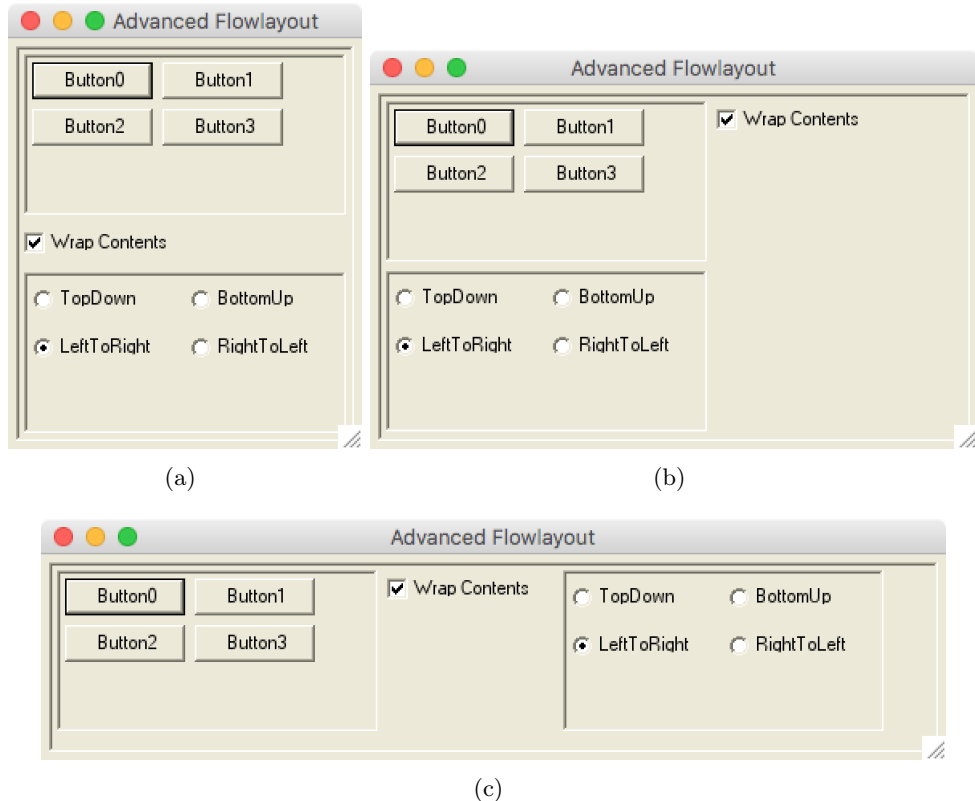
Figure 24.17: Nested `FlowLayoutPanel`, see **??**–**??**, allows for dynamic arrangement
of content. Content flows when the window is resized.

Listing 24.20 winforms/flowLayoutPanel.fsx:
Create a FlowLayoutPanel with checkbox and radio buttons.

```fsharp
open System.Windows.Forms
open System.Drawing

let flowLayoutPanel = new FlowLayoutPanel ()
let buttonLst =
  [(new Button (), "Button0");
   (new Button (), "Button1");
   (new Button (), "Button2");
   (new Button (), "Button3")]
let panel = new Panel ()
let wrapContentsCheckBox = new CheckBox ()
let initiallyWrapped = true
let radioButtonLst =
  [(new RadioButton (), (3, 34), "TopDown",
    FlowDirection.TopDown);
   (new RadioButton (), (3, 62), "BottomUp",
    FlowDirection.BottomUp);
   (new RadioButton (), (111, 34), "LeftToRight",
    FlowDirection.LeftToRight);
   (new RadioButton (), (111, 62), "RightToLeft",
    FlowDirection.RightToLeft)]

// customize buttons
for (btn, txt) in buttonLst do
  btn.Text <- txt

// customize wrapContentsCheckBox
wrapContentsCheckBox.Location <- new Point (3, 3)
wrapContentsCheckBox.Text <- "Wrap Contents"
wrapContentsCheckBox.Checked <- initiallyWrapped
wrapContentsCheckBox.CheckedChanged.Add (fun _ ->
    flowLayoutPanel.WrapContents <- wrapContentsCheckBox.Checked)

// customize radio buttons
for (btn, loc, txt, dir) in radioButtonLst do
  btn.Location <- new Point (fst loc, snd loc)
  btn.Text <- txt
  btn.Checked <- flowLayoutPanel.FlowDirection = dir
  btn.CheckedChanged.Add (fun _ -> flowLayoutPanel.FlowDirection
   <- dir)
```

Listing 24.21 winforms/flowLayoutPanel.fsx:
Create a FlowLayoutPanel with checkbox and radio buttons. Contin-
ued from ??.

```
36  // customize flowLayoutPanel
37  for (btn, txt) in buttonLst do
38    flowLayoutPanel.Controls.Add btn
39  flowLayoutPanel.Location <- new Point (47, 55)
40  flowLayoutPanel.BorderStyle <- BorderStyle.Fixed3D
41  flowLayoutPanel.WrapContents <- initiallyWrapped
42
43  // customize panel
44  panel.Controls.Add (wrapContentsCheckBox)
45  for (btn, loc, txt, dir) in radioButtonLst do
46    panel.Controls.Add (btn)
47  panel.Location <- new Point (47, 190)
48  panel.BorderStyle <- BorderStyle.Fixed3D
49
50  // Create a window, add controls, and start event-loop
51  let win = new Form ()
52  win.ClientSize <- new Size (302, 356)
53  win.Controls.Add flowLayoutPanel
54  win.Controls.Add panel
55  win.Text <- "A Flowlayout Example"
56  Application.Run win
```

Listing 24.22 winforms/flowLayoutPanelAdvanced.fsx: Create nested FlowLayoutPanel.

```fsharp
open System.Windows.Forms
open System.Drawing
open System

let win = new Form ()
let mainPanel = new FlowLayoutPanel ()
let mainPanelBorder = 5
let flowLayoutPanel = new FlowLayoutPanel ()
let buttonLst =
  [(new Button (), "Button0");
   (new Button (), "Button1");
   (new Button (), "Button2");
   (new Button (), "Button3")]
let wrapContentsCheckBox = new CheckBox ()
let panel = new FlowLayoutPanel ()
let initiallyWrapped = true
let radioButtonLst =
  [(new RadioButton (), "TopDown", FlowDirection.TopDown);
   (new RadioButton (), "BottomUp", FlowDirection.BottomUp);
   (new RadioButton (), "LeftToRight", FlowDirection.LeftToRight);
   (new RadioButton (), "RightToLeft", FlowDirection.RightToLeft)]

// customize buttons
for (btn, txt) in buttonLst do
  btn.Text <- txt

// customize radio buttons
for (btn, txt, dir) in radioButtonLst do
  btn.Text <- txt
  btn.Checked <- flowLayoutPanel.FlowDirection = dir
  btn.CheckedChanged.Add (fun _ -> flowLayoutPanel.FlowDirection
   <- dir)

// customize flowLayoutPanel
for (btn, txt) in buttonLst do
  flowLayoutPanel.Controls.Add btn
flowLayoutPanel.BorderStyle <- BorderStyle.Fixed3D
flowLayoutPanel.WrapContents <- initiallyWrapped

// customize wrapContentsCheckBox
wrapContentsCheckBox.Text <- "Wrap Contents"
wrapContentsCheckBox.Checked <- initiallyWrapped
wrapContentsCheckBox.CheckedChanged.Add (fun _ ->
    flowLayoutPanel.WrapContents <- wrapContentsCheckBox.Checked)
```

Listing 24.23 winforms/flowLayoutPanelAdvanced.fsx: Create nested FlowLayoutPanel. Continued from ??.

```
44  // customize panel
45  // changing border style changes ClientSize
46  panel.BorderStyle <- BorderStyle.Fixed3D
47  let width = panel.ClientSize.Width / 2 - panel.Margin.Left -
      panel.Margin.Right
48  for (btn, txt, dir) in radioButtonLst do
49    btn.Width <- width
50    panel.Controls.Add (btn)
51
52  mainPanel.Location <- new Point (mainPanelBorder, mainPanelBorder)
53  mainPanel.BorderStyle <- BorderStyle.Fixed3D
54  mainPanel.Controls.Add flowLayoutPanel
55  mainPanel.Controls.Add wrapContentsCheckBox
56  mainPanel.Controls.Add panel
57
58  // customize window, add controls, and start event-loop
59  win.ClientSize <- new Size (220, 256)
60  let windowResize _ =
61    let size = win.DisplayRectangle.Size
62    mainPanel.Size <- new Size (size.Width - 2 * mainPanelBorder,
      size.Height - 2 * mainPanelBorder)
63  windowResize ()
64  win.Resize.Add windowResize
65  win.Controls.Add mainPanel
66  win.Text <- "Advanced Flowlayout"
67  Application.Run win
```

25 The Event-driven Programming Paradigm

In *event-driven programming*, the flow of the program is determined by *events*, such as the user moving the mouse, an alarm going off, a message arriving from another program, or an exception being thrown, and is very common for programs with extensive interaction with a user, such as a graphical user interface. The events are monitored by *listeners*, and the programmer can set *handlers* which are *call-back functions* to be executed when an event occurs. In event-driven programs, there is almost always a main loop to which the program relinquishes control to when all handlers have been set up. Event-driven programs can be difficult to test, since they often rely on difficult-to-automate mechanisms for triggering events, e.g., testing a graphical user interface often requires users to point-and-click, which is very slow compared to automatic unit testing.

26 Where to Go from Here

You have now learned to program in a number of important paradigms and mastered the basics of F#, so where are good places to go now? I will highlight a number of options:

Program, program, program
You are at this stage no longer a novice programmer, so it is time for you to use your skills and create programs that solve problems. I have always found great inspiration in interacting with other domains and seeking solutions by programming. Experience is a must if you want to become a good programmer, since your newly acquired skills need to settle in your mind, and you need to be exposed to new problems that require you to adapt and develop your skills.

Learn to use an Integrated Development Environment effectively
An Integrated Development Environment (IDE) is a tool that may increase your coding efficiency. IDEs can help you get started in different environments, such as on a laptop or a phone, and it can quickly give you an overview of available options when you are programming. E.g., all IDEs will show you available members for identifiers as you type, reducing time to search members and reducing the risk of spelling errors. Many IDEs will also help you to quickly refactor your code, e.g., by highlighting all occurrences of a name in a scope and letting you change all of them in one action.

In this book, we have emphasized the console. Compiling and running from the console is the basis of which all IDEs build, and many of the problems with using IDEs efficiently are related to understanding how it can best help you compiling and running programs.

Learn other cool features of F#
F# is a large language with many features. Some have been presented in this book, but more advanced topics have been neglected. Examples are:

- regular expressions: Much computations concern processing of text. Regular expressions is a simple but powerful language for searching and replacing in strings. F# has built-in support for regular expressions as `System.Text.RegularExpressions`.

- sequence `seq`: All list type data structures in F# are built on sequences. Sequences are, however, more than lists and arrays. A key feature is that sequences can effectively contain large or even infinite ordered lists which you do not necessarily need or use, i.e., they are lazy and only compute its elements as needed. Sequences are programmed using computation expressions.

- computation expressions: Sequential expressions is an example of computation expressions, e.g., the sequence of squares $i^2, i = 0..9$ can be written as `seq {for i in 0 .. 9 -> i * i}`

- asynchronous computations `async`: F# has a native implementation of asynchronous computation, which means that you can very easily set up computations that run independently of others, such that they do not block each other. This is extremely convenient if you, e.g., need to process a list of homepages, where each homepage may be slow to read, such that reading them in sequence will be slow. With asynchronous computations, they can easily be read in parallel with a huge speedup for the total task as a result. Asynchronous workflows rely on computation expressions.

Learn another programming language

F# is just one of a great number of programming languages, and you should not limit yourself. Languages are often designed to be used for particular tasks, and when looking to solve a problem, you would do well in selecting the language that best fits the task. C# is an example from the Mono family which emphasizes object-oriented programming, and many of the built-in libraries in F# are written in C#. C++ and C are ancestors of C# and are very popular since they allow for great control over the computer at the expense of programming convenience. Python is a popular prototyping language which emphasizes interactive programming like `fsharpi`, and it is seeing a growing usage in web-backends and machine learning. And the list goes on. To get an idea of the wealth of languages, browse `http://www.99-bottles-of-beer.net` which has examples of solutions to the simple problem: Write a program that types the lyrics of song "99 bottles of beer on the wall" and stop. At the present time, many solutions in more than 1500 different languages have been submitted.

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 **??**.

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 **??**.

(a) Linux

(b) MacOS X
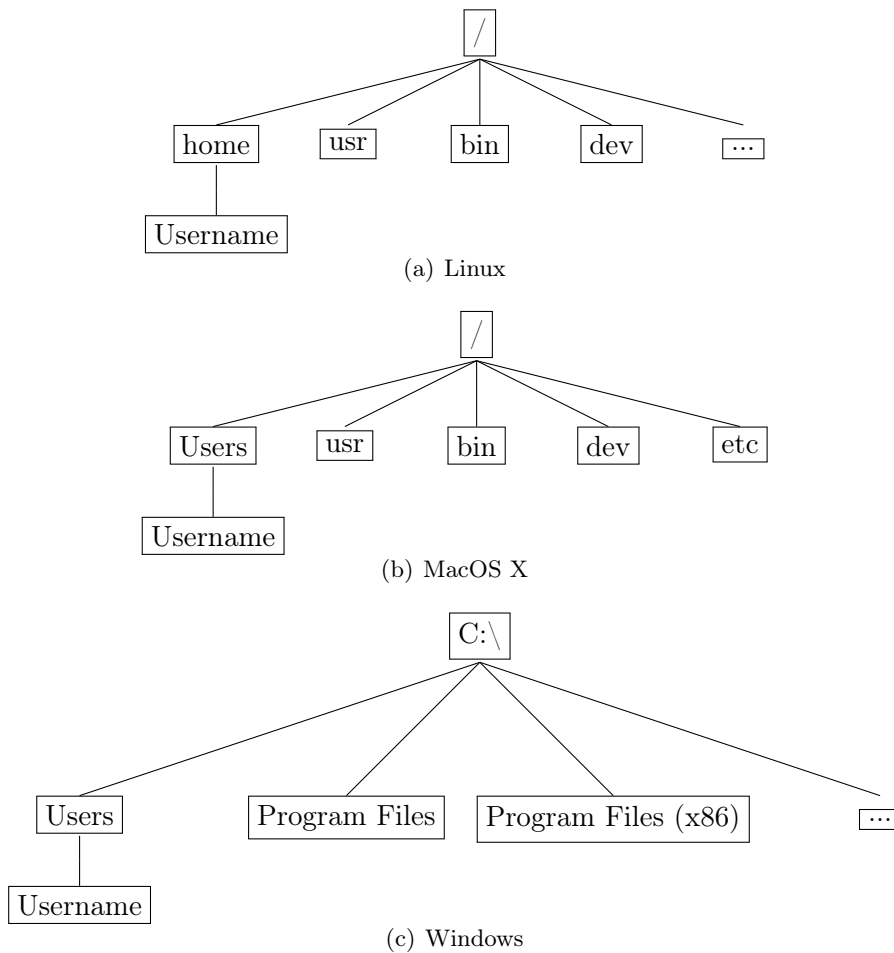
(c) Windows

Figure A.1: The top file hierarchy levels of common operating systems.

A.2 Windows

In this section we will discuss the commands summarized in **??**. 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 **??**.

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.

Listing A.1: The Windows console.

```
1  Microsoft Windows [Version 6.1.7601]
2  Copyright (c) 2009 Microsoft Corporation.  All rights
      reserved.
3
4  C:\Users\sporring>
```

To see which files are in the directory, use *dir*, as shown in **??**.

Listing A.2: Directory listing with dir.

```
1   C:\Users\sporring>dir
2    Volume in drive C has no label.
3    Volume Serial Number is 94F0-31BD
4
5    Directory of C:\Users\sporring
6
7   30-07-2015  15:23    <DIR>          .
8   30-07-2015  15:23    <DIR>          ..
9   30-07-2015  14:27    <DIR>          Contacts
10  30-07-2015  14:27    <DIR>          Desktop
11  30-07-2015  17:40    <DIR>          Documents
12  30-07-2015  15:11    <DIR>          Downloads
13  30-07-2015  14:28    <DIR>          Favorites
14  30-07-2015  14:27    <DIR>          Links
15  30-07-2015  14:27    <DIR>          Music
16  30-07-2015  14:27    <DIR>          Pictures
17  30-07-2015  14:27    <DIR>          Saved Games
18  30-07-2015  17:27    <DIR>          Searches
19  30-07-2015  14:27    <DIR>          Videos
20                 0 File(s)              0 bytes
21                13 Dir(s)  95.004.622.848 bytes free
22
23  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: Change directory with cd.

```
1   C:\Users\sporring>cd Documents
2
3   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: Creating a directory with `mkdir`.

```
1   C:\Users\sporring\Documents>mkdir myFolder
2
3   C:\Users\sporring\Documents>dir
4    Volume in drive C has no label.
5    Volume Serial Number is 94F0-31BD
6
7    Directory of C:\Users\sporring\Documents
8
9   30-07-2015  19:17    <DIR>              .
10  30-07-2015  19:17    <DIR>              ..
11  30-07-2015  19:17    <DIR>              myFolder
12               0 File(s)              0 bytes
13               3 Dir(s)  94.656.638.976 bytes free
14
15  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: Creating a file with `echo` and redirection.

```
1   C:\Users\sporring\Documents>echo "Hi" > hi.txt
2
3   C:\Users\sporring\Documents>dir
4    Volume in drive C has no label.
5    Volume Serial Number is 94F0-31BD
6
7    Directory of C:\Users\sporring\Documents
8
9   30-07-2015  19:18    <DIR>              .
10  30-07-2015  19:18    <DIR>              ..
11  30-07-2015  19:17    <DIR>              myFolder
12  30-07-2015  19:18                   8 hi.txt
13               1 File(s)              8 bytes
14               3 Dir(s)  94.656.634.880 bytes free
15
16  C:\Users\sporring\Documents>
```

To move the file `hi.txt` to the directory `myFolder`, use *move*, as shown in **??**.

> **Listing A.6: Move a file with `move`.**
>
> ```
> 1  C:\Users\sporring\Documents>move hi.txt myFolder
> 2          1 file(s) moved.
> 3
> 4  C:\Users\sporring\Documents>
> ```

Finally, use *del* to delete a file and *rmdir* to delete a directory, as shown in **??**.

> **Listing A.7: Delete files and directories with `del` and `rmdir`.**
>
> ```
> 1  C:\Users\sporring\Documents>cd myFolder
> 2
> 3  C:\Users\sporring\Documents\myFolder>del hi.txt
> 4
> 5  C:\Users\sporring\Documents\myFolder>cd ..
> 6
> 7  C:\Users\sporring\Documents>rmdir myFolder
> 8
> 9  C:\Users\sporring\Documents>dir
> 10   Volume in drive C has no label.
> 11   Volume Serial Number is 94F0-31BD
> 12
> 13   Directory of C:\Users\sporring\Documents
> 14
> 15  30-07-2015  19:20    <DIR>          .
> 16  30-07-2015  19:20    <DIR>          ..
> 17              0 File(s)              0 bytes
> 18              2 Dir(s)  94.651.142.144 bytes free
> 19
> 20  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: Displaying the search path.**
>
> ```
> 1  C:\Users\sporring\Documents>echo %Path%
> 2  C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;
>       C:\Windows\System32\WindowsPowerShell\v1.0\;"\Program
>       Files\emacs-24.5\bin\"
> 3
> 4  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 **??**. 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: The MacOS console.

```
1  Last login: Thu Jul 30 11:52:07 on ttys000
2  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: Display a directory content with `ls`.

```
1  FN11194:~ sporring$ ls
2  Applications    Documents    Library      Music
     Public
3  Desktop         Downloads    Movies       Pictures
4  FN11194:~ sporring$
```

More details about the files are available by using flags to `ls` as demonstrated in **??**.

Listing A.11: Display extra information about files using flags to `ls`.

```
1  FN11194:~ sporring$ ls -l
2  drwx------   6 sporring   staff    204 Jul 30 14:07
     Applications
3  drwx------+ 32 sporring   staff   1088 Jul 30 14:34 Desktop
4  drwx------+ 76 sporring   staff   2584 Jul  2 15:53 Documents
5  drwx------+  4 sporring   staff    136 Jul 30 14:35 Downloads
6  drwx------@ 63 sporring   staff   2142 Jul 30 14:07 Library
7  drwx------+  3 sporring   staff    102 Jun 29 21:48 Movies
8  drwx------+  4 sporring   staff    136 Jul  4 17:40 Music
9  drwx------+  3 sporring   staff    102 Jun 29 21:48 Pictures
10 drwxr-xr-x+  5 sporring   staff    170 Jun 29 21:48 Public
11 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: Change directory with `cd`.

```
1  FN11194:~ sporring$ cd Documents/
2  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: Creating a directory using `mkdir`.

```
1  FN11194: Documents  sporring$   mkdir myFolder
2  FN11194: Documents  sporring$ ls
3  myFolder
4  FN11194: tmp  sporring$
```

A file can be created using **`echo`** and with *redirection*, as shown in **??**.

Listing A.14: Creating a file with `echo` and redirection.

```
1  FN11194: Documents  sporring$ echo "hi" > hi.txt
2  FN11194: Documents  sporring$ ls
3  hi.txt          myFolder
```

To move the file `hi.txt` into `myFolder`, use **`mv`**. This is demonstrated in **??**.

Listing A.15: Moving files with `mv`.

```
1  FN11194: Documents  sporring$ echo mv hi.txt myFolder/
2  FN11194: Documents  sporring$
```

To delete the file and the directory, use **`rm`** and **`rmdir`**, as shown in **??**.

Listing A.16: Deleting files and directories.

```
1  FN11194: Documents  sporring$ cd myFolder/
2  FN11194: myFolder  sporring$ rm hi.txt
3  FN11194: myFolder  sporring$ cd ..
4  FN11194: Documents  sporring$ rmdir myFolder/
5  FN11194: Documents  sporring$ ls
6  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: The content of the search path.

```
1  FN11194:Documents sporring$ echo $PATH
2  /Applications/Maple
      17/:/Applications/PackageMaker.app/Contents/MacOS/:
      /Applications/MATLAB_R2014b.app/bin/:/opt/local/bin:
      /opt/local/sbin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:
      /sbin:/opt/X11/bin:/Library/TeX/texbin
3  FN11194:Documents sporring$
```

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.

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:

$$v = \sum_{i=-m}^{n} d_i 10^i. \tag{B.19}$$

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,

$$v = \sum_{i=-m}^{n} b_i 2^i, \tag{B.20}$$

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 **??**.

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,

$$e = \sum_{i=1}^{11} e_i 2^{11-i}, \tag{B.21}$$

$$m = \sum_{j=1}^{52} m_j 2^{-j}. \tag{B.22}$$

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,

$$v = (-1)^s \, (1+m)\, 2^{e-1023} \tag{B.23}$$

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)
$e = 2^{11}-1$	$v = (-1)^s\, \infty$	$v = (-1)^s\, \text{NaN}$ (not-a-number)

where $e = 2^{11} - 1 = 11111111111_2 = 2047$. The largest and smallest number that is not infinity is thus

$$e = 2^{11} - 2 = 2046, \tag{B.24}$$

$$m = \sum_{j=1}^{52} 2^{-j} = 1 - 2^{-52} \simeq 1, \tag{B.25}$$

$$v_{\max} = \pm \left(2 - 2^{-52}\right) 2^{1023} \simeq \pm 2^{1024} \simeq \pm 10^{308}. \tag{B.26}$$

The density of numbers varies in such a way that when $e - 1023 = 52$, then

$$v = (-1)^s \left(1 + \sum_{j=1}^{52} m_j 2^{-j} \right) 2^{52} \tag{B.27}$$

$$= \pm \left(2^{52} + \sum_{j=1}^{52} m_j 2^{-j} 2^{52} \right) \tag{B.28}$$

$$= \pm \left(2^{52} + \sum_{j=1}^{52} m_j 2^{52-j} \right) \tag{B.29}$$

$$\overset{k=52-j}{=} \pm \left(2^{52} + \sum_{k=51}^{0} m_{52-k} 2^k \right), \tag{B.30}$$

which are all integers in the range $2^{52} \leq |v| < 2^{53}$. When $e - 1023 = 53$, then the same calculation gives

$$v \overset{k=53-j}{=} \pm \left(2^{53} + \sum_{k=52}^{1} m_{53-k} 2^k \right), \tag{B.31}$$

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,

$$v \overset{k=51-j}{=} \pm \left(2^{51} + \sum_{k=50}^{-1} m_{51-k} 2^k \right), \tag{B.32}$$

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

$$v = (-1)^s \left(\sum_{j=1}^{52} m_j 2^{-j} \right) 2^{-1022} \tag{B.33}$$

$$= \pm \left(\sum_{j=1}^{52} m_j 2^{-j} 2^{-1022} \right) \tag{B.34}$$

$$= \pm \left(\sum_{j=1}^{52} m_j 2^{-j-1022} \right) \tag{B.35}$$

$$\overset{k=-j-1022}{=} \pm \left(\sum_{j=-1023}^{-1074} m_{-k-1022} 2^k \right), \tag{B.36}$$

which gives a distance between numbers of $2^{-1074} \simeq 10^{-323}$ in the range $0 < |v| < 2^{-1022} \simeq 10^{-308}$.

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*) [**?**], is a 7 bit code tuned for the letters of the English language, numbers, punctuation symbols, control codes and space, see **????**. The first 32 codes are reserved for 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

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

'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. **??** shows the characters above 7e. Codes 00-1f and 7f-9f are undefined in ISO 8859-1.

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

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.

its block of the first 128 code points is called the *Basic Latin block* and is identical to ASCII, see **??**, and code points 128-255 are called the *Latin-1 Supplement block*, and are identical to the upper range of ISO 8859-1, see **??**. 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 ob-

x0+0x	80	90	A0	B0	C0	D0	E0	F0
00			NBSP	°	À	Đ	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.

tained 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 ??.

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-

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.

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.

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 [**?, ?**]. 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 **??**. 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 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.
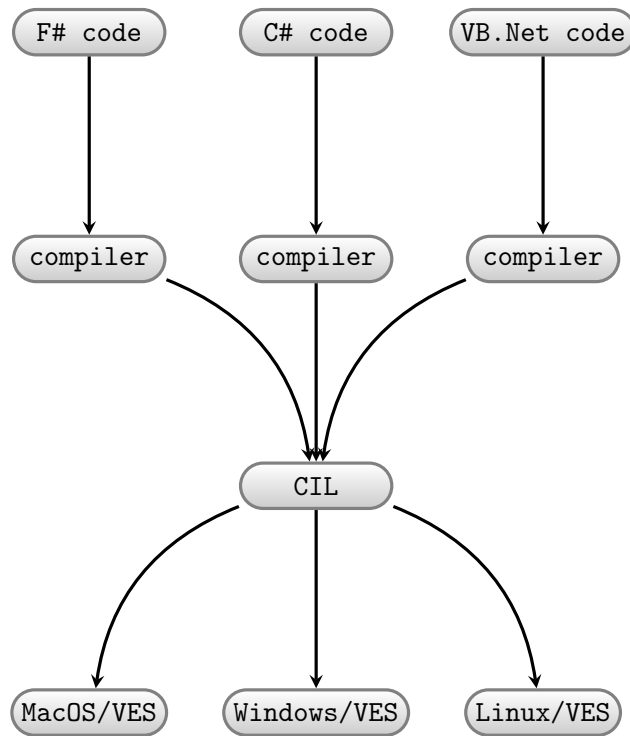
Figure D.1: The relation between some .NET/Mono languages with the Common intermediate language (CIL), and the Virtual execution systems (VES) on some operating system (`mono`).

Bibliography

[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