Learning to program with F#

Jon Sporring

Department of Computer Science,
University of Copenhagen

November 19, 2017

Contents

1 Preface

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

Jon Sporring
Associate Professor, Ph.d.
Department of Computer Science,
University of Copenhagen
November 19, 2017

2 Introduction

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

2.1 Learning how to solve problems by programming

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

1. Choose a programming language: A programming language such as F# is a vocabulary and a set of gramatical rules for instructing a computer to perform a certain task. It is possible to program without a concrete language, but your ideas and thoughts must be expressed in some fairly rigorous way. Actually, theoretical computer science typically does not rely on computers nor programming languages, but uses mathematics to prove properties of algorithms. However, most computer scientists program, and with a real language, you have the added benefit of checking your algorithm, and hence your thoughts, rigorously on a real computer. This book teaches a subset of F#. The purpose is not to be a reference guide to this language, but to use it as a vessel to teach you, the reader, how to convert your ideas into programs.

2. Learn the language: A computer language is a structure for thought, and it influences which thoughts you choose to implement as a program, and how you choose to do it. Any conversion requires you to acquire a sufficient level of fluency, for you to be able to make programs. You do not need to be a master in F# nor to know every corner of the language, and you will expand your knowledge as you expose yourself to solving problems in the language, but you must invest an initial amount of time and energy in order to learn the basics of the language. This book aims at getting you started quickly, which is why we intentionally teach just a small subset of F#. On the net and through other works, you will be able to learn much more.

3. Practice: If you want to be a good programmer, then there is only one way: practice, practice, practice! It has been estimated that to master anything, then you have to have spent at least 10000 hours of practice, so get started logging hours! It of course matters, what you practice. This book teaches 3 different programming themes. The point is that programming is thinking, and the scaffold that you use, shapes your thoughts. It is therefore important to recognise this scaffold, and to have the ability to choose that which suits your ideas and your goals best. And the best way to expand your abilities is to sharpen your present abilities, push yourself into new territory, and trying something new. Do not be afraid to make errors or be frustrated at first. These are the experiences that make you grow.

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

2.2 How to solve problems

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

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

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

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

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

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

2.3 Approaches to programming

This book focuses on 3 fundamentally different approaches to programming:

· imperative programming

Imperative programming, emphasises *how a program shall accomplish a solution* and less on *what the solution is*. A cooking recipe is an example of the spirit of imperative programming, where the recipe emphasises what should be done in each step rather than describing the result. E.g., for making bread, you first mix yeast and water, then add flour, etc. In imperative programming

· statement
· state

what should be done are called *statements* and they influence the computer's *states*, like adding flour changes the state of our dough. Almost all computer hardware is designed to execute low-level programs written in imperative style. Imperative programming builds on the Turing machine [10]. The first major language was FORTRAN [6] which emphasized an imperative style of programming.

Declarative programming, which emphasises *what a program shall accomplish* but not *how*. We will consider Functional programming as an example of declarative programming. A *functional programming* language evaluates *functions* and avoids state changes. The program consists of *expressions* instead of statements. As an example the function $f(x) = x^2$ takes a number x and evaluates the expression x^2, and returns the result. Functional programming has its roots in lambda calculus [1], and the first language emphasizing functional programming was Lisp [7].

· declarative programming
· functional programming
· function
· expression

Structured programming, which emphasises organisation of code in units with well-defined interfaces and isolation of internal states and code from other parts of the program. We will focus on Object-oriented programming as the example of structured programming. *Object-oriented programming* is a type of programming, where the states and programs are structured into *objects*. A typical object-oriented design takes a problem formulation and identifies key nouns as potential objects and verbs as potential actions to be taken on objects. The first object-oriented programming language was Simula 67 developed by Dahl and Nygaard at the Norwegian Computing Center in Oslo [2].

· structured programming
· Object-oriented programming
· object

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

· event-driven programming
· call-back functions

Most programs do not follow a single programming paradigm as, e.g., one of the above, but are a mix. Nevertheless, this book will treat each paradigm separately to emphasize their advantages and disadvantages.

2.4 Why use F#

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

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

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

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

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

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

Assemblies: F# is designed to be able to easily communicate with other .Net and Mono programs through the language-independent, platform-independent bytecode called Common Intermediate Language (CIL) organised as assemblies. Thus, if you find that certain parts of a program are easy to express in F# and others in C++, then you will be able to combine these parts later into a single program.

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

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

2.5 How to read this book

Learning to program requires mastering a programming language, however most programming languages contains details that are rarely used or used in contexts far from a specific programming topic. Hence, this book only includes a subset of F#, but focuses on language structures necessary to understand 4 common programming paradigms: Imperative programming mainly covered in Chapters 6 to 11, functional programming mainly covered in Chapters 13 to 16, object oriented programming in Chapters 20 and 21, and event driven programming in Chapter 22. A number of general topics are given in the appendix for reference. The disadvantage of this approach is that no single part contains a reference guide to F#, and F# topics are revisited and expanded across the book. For further reading please consult `http://fsharp.org`.

3 Executing F# code

3.1 Source code

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

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

· interactive mode
· compile mode
· console

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

· source code

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

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

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

`.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` · library file

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

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

· scripts
· script-fragment

3.2 Executing programs

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

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

Listing 3.1 gettingStartedStump.fsx:
A simple demonstration script.

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

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

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

Listing 3.2: An interactive session.

```
$ fsharpi

F# Interactive for F# 4.1 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License

For help type #help;;

> let a = 3.0
- do printfn "%g" a;;
3

val a : float = 3.0
val it : unit = ()

> #quit;;
```

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

[1]Jon: **Too early to introduce lexeme: "F# uses many characters which at times are given special meanings, e.g., the characters "`;;`" is compound character denoting end of a script-fragment. Such possibly compound characters are called lexemes."**

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

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

Listing 3.3: Using the interpreter to execute a script.

```
$ fsharpi gettingStartedStump.fsx
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.

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

Listing 3.4: Compiling and executing a script.

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

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

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

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`, $1.88\text{s} < 0.05\text{s} + 1.90\text{s}$, 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`, $1.88\text{s} \gg 0.05\text{s}$.

The interactive session results in extra output on the *type inference* performed, which is very useful for *debugging* and development of code-fragments, but both executing programs with the interpreted directly from a file and compiling and executing the program is much preferred for programming complete programs, since the starting state is well defined, and since this better supports *unit-testing*, which is a method for debugging programs. Thus, **prefer compiling over interpretation.**

· type inference
· debugging
· unit-testing
Advice

4 | Quick-start guide

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

Problem 4.1

What is the sum of 357 and 864?

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

Listing 4.1 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
```

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

To solve the problem, we made program consisting of several lines, where each line was a *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 a let-binding makes the name synonymous with the value. Another point to be noted is that F# identifies 357 as an *integer number*, which is F#'s preferred number type, since computations on integers are very efficient, and since integers are very easy to communicate to other programs. In line 2 we bound the value 864 to the name `b`, and to the name `c`, we bound the result of evaluating the sum `a + b` in line 3. Line 4 is a *do-binding*, as noted by the keyword *`do`*. Do-bindings are also sometimes called *statements*, and the `do` keyword is optional in F#. Here the value of `c` was printed to the console followed by a newline (LF possibly preceded by CR, see Appendix C.1) with the *`printfn` function*. A function in F# is an entity that takes zero or more arguments and returns a value. The function `printfn` is very special, since it can take any number of arguments. It need not return any value, but F# insists that every function must return a value, wherefore `printfn`

· expression
· `let`
· keyword
· binding
· let-binding
· integer number
· do-binding
· `do`
· statements
· `printfn`
· function

returns a special type of value called ***unit*** and written as *"()"*. The `do` tells F# to ignore this value. Here `printfn` has been used with 2 arguments: `"%A"` and `c`. Notice that in contrast to many other languages, F# does not use parentheses to frame the list of arguments, nor does it use commas to separate them. In general, the `printfn` function always has 1 or more arguments, and the first is a *format string*. A *string* is a sequence of characters starting and ending with double quotation marks. E.g., `let s = "this is a string of characters"` binds the string `"this is..."` to the name `s`. For the `printfn` function, the format string may be any string, but if it contains format character sequences, such as `%A`, then the values following the format string are substituted. The format string must match the value *type*, that is, here `c` is of type integer, whereas the format string `%A` matches many types.

· unit
· ()
· format string
· string
· type

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

· type declaration
· type inference

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

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

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

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

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

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

· val
Advice
· it
· ()

Were we to solve a slightly different problem,

Problem 4.2

What is the sum of 357.6 and 863.4?

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

· decimal point
· floating point numbers

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

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

```
$ fsharpc --nologo quickStartSumFloat.fsx && mono quickStartSumFloat.exe
1221.0
```

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

Listing 4.4: Mixing types is often not allowed.

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

> let b = 863.4;;
val b : float = 863.4

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

stdin(4,13): error FS0001: The type 'float' does not match the type 'int'
```

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

· error message

· debugging

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

· lexical scope

Listing 4.5 quickStartRebindError.fsx:
A name cannot be rebound.

```
let a = 357
let a = 864
```

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

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

However, if the same is performed in an interactive session, then rebinding does not cause an error as shown in Listing 4.6.

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

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

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

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

· ;;
· lexeme
· script-fragment
Advice

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

· function

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

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

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

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

· encapsulate

Type inference in F# may cause problems, since the type of a function is inferred in the context, in which it is defined. E.g., in an interactive session, defining the sum in one scope on a single line will default the types to integers, F#'s favorite type. Thus, if the next script-fragment uses the function with floats, then we will get an error message as shown in Listing 4.8.

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

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

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

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

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

Listing 4.9: Type inference is per script-fragment.

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

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

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

```
> let sum (x : float) (y : float) : float = x + y;;
val sum : x:float -> y:float -> float

> let c = sum 357.6 863.4;;
val c : float = 1221.0
```

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

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 Using F# as a calculator

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

5.1 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 Listing 5.1.

· type
· literal

Listing 5.1: Typing the number 3.

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

What this means is that F# has inferred the type to be ***int*** and bound it to the identifier ***it***. For more on binding and identifiers see Chapter 6. Types matter, since the operations that can be performed on integers are quite different from those that can be performed on, e.g., strings. E.g., the number 3 has many different representations as shown in Listing 5.2.

· `int`
· `it`

Listing 5.2: Many representations of the number 3 but using different types.

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

> 3.0;;
val it : float = 3.0

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

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

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 Table 5.1. Besides these built-in types, F# is designed such that it is easy to define new types.

· `float`
· `bool`
· `char`
· `string`
· basic types

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

Table 5.1: List of some of the basic types. The most commonly used types are underlined. For at description of integer see Appendix B.1, for floating point numbers see Appendix B.2, for ASCII and Unicode characters see Appendix C, for objects see Chapter 20, and for exceptions see Chapter 18.

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. The whole part with neither a decimal point nor a fractional part is called an *integer*. As an example `35.7` is a decimal number, whose value is $3 \cdot 10^1 + 5 \cdot 10^0 + 7 \cdot 10^{-1}$, and `128` is an integer, whose value is $1 \cdot 10^2 + 2 \cdot 10^1 + 8 \cdot 10^0$. In F# a decimal number is called a *floating point number*. Floating point numbers may alternatively be given using *scientific notation*, such as `3.5e-4` and `4e2`, where the `e`-notation is translated to a value as `3.5e-4` $= 3.5 \cdot 10^{-4} = 0.00035$, and `4e2` $= 4 \cdot 10^2 = 400$.

· decimal number
· base
· decimal point
· digit
· whole part
· fractional part
· integer
· floating point number
· scientific notation

The basic unit of information in almost all computers is the binary digit or *bit* for short. Internally, programs and data is 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.

· bit
· binary number

Binary numbers are closely related to *octal* and *hexadecimal numbers*, where octals uses 8 as basis, and where each octal digit can be represented by exactly 3 bits, while hexadecimal numbers uses 16 as basis, and where each hexadecimal digit can be written in binary using exactly 4 bits. The hexadecimal digits uses `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 $15 \cdot 16^1 + 3 \cdot 16^0 = 243$.

· octal number
· hexadecimal number

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

Table 5.2: Escape characters. For the unicode characters ’X’ are hexadecimal digits, while for tricode characters ’D’ is a decimal character.

To denote integers on 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`. The character sequences `0b12` and `ff` are not numbers.

A *character* is a *Unicode code point*, and character literals are enclosed in single quotation marks. Appendix C.3 contains more details on code points. The character type in F# is denoted *`char`*. Examples of characters are ’a’, ’D’, ’3’. However ’23’ and ’abc’ are not characters. Some characters do not have a visual representation such as the tabulation character. These can still be represented as a character using *escape sequences*. A character escape sequence starts with “\” followed by either a letter for simple escapes such as `\t` for tabulation and `\n` for newline. Escape sequences can also be a numerical representation of a code point, and three versions exist: The trigraph `\DDD`, where `D` is a decimal digit, is used to specify the first 256 code points, the hexadecimal escape codes `\uXXXX`, where `X` is a hexadecimal digit, is used to specify the first 65536 code points, and `\UXXXXXXXX` is used to specify any of the approximately $4.3 \cdot 10^9$ possible code points. All escape sequences are shown in Table 5.2. Examples of `char` representations of the letter ’a’ are: `'a'`, `'\097'`, `'\u0061'`, `'\U00000061'`.

· character
· Unicode
· code point
· `char`
· escape sequences

A *string* is a sequence of characters enclosed in double quotation marks. Examples are `"a"`, `"this is a string"`, and `"-&#\@"`. Note that the string `"a"` and the character `'a'` are not the same. Some strings are so common that they are given special names: One or more spaces, `" "` is called *whitespace*, and both `"\n"` and `"\r\n"` are called *newline*. The escape-character “\” may be used to break a line in two. This and other examples are shown in Listing 5.3.

· string
· whitespace
· newline

Type	syntax	Examples	Value
int, int32	<int or hex> <int or hex>l	3, 0x3 3l, 0x3l	3
uint32	<int or hex>u <int or hex>ul	3u 3ul	3
byte, uint8	<int or hex>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 or hex>y	3y	3
int16	<int or hex>s	3s	3
uint16	<int or hex>us	3us	3
int64	<int or hex>L	3L	3
uint64	<int or hex>UL <int or hex>uL	3UL 3uL	3
float, double	<float> <hex>LF	3.0 0x013fLF	3.0 9.387247271e-323
single, float32	<float>F <float>f <hex>lf	3.0F 3.0f 0x013flf	3.0 3.0 4.4701421e-43f
decimal	<float or int>M <float or int>m	3.0M,3M 3.0m,3m	3.0
string	"<string>" @"<string>" ""<string>""	"a \"quote\".\n" @"a ""quote"".\n" """a "quote".\n"""	a "quote".<newline> a "quote".\n. a "quote".\n

Table 5.3: List of literal type. Syntax notation is used such that, e.g., <> means that the programmer replaces the brackets and content with a value on appropriate form. The [||] notation means that the value is an array, see Section 11.3 for details.

Listing 5.3: Examples of string literals.

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

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

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

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

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 og suffix as shown in the Table 5.3. The table uses a simple syntax notation such that <integer or hexadecimal>UL means that the user supplies an integer or a hexadecimal number followed by the characters 'UL'.

· literal type

The literal type is closely connected to how the values are represented internally. E.g., 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` uses 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 Table 5.1 and discussed in more detail in Appendix B. String literals may be *verbatim* by the @-notation or triple double quotation marks, meaning that the escape sequences are not converted to their code point. The two types of string verbatim treat quotation marks differently as illustrated in the table. Further examples are shown in Listing 5.4.

· verbatim

Listing 5.4: Named and implied literals.

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

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

> 5.6;;
val it : float = 5.6

> 7.9f;;
val it : float32 = 7.9000001f

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

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

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

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

Many basic types are compatible, and the type of a literal may be changed by *typecasting*. An example of casting to a `float` is shown in Listing 5.5.

· typecasting

Listing 5.5: Casting an integer to a floating point number.

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

which is a `float`, since 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 Chapter 6. Boolean values are often treated as the integer values 0 and 1, but no short-hand function names exists for their conversions. Instead use functions from the `System.Convert` family of functions, as demonstrated in Listing 5.6.

Listing 5.6: Casting booleans.

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

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

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

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

Here `System.Convert.ToBoolean` is the identifier of a function `ToBoolean`, which is a *member* of the *class* `Convert` that is included in the *namespace* `System`. Namespaces, classes, and members will be discussed in Chapter 9.

· member
· class
· namespace

Typecasting is often a destructive operation, e.g., typecasting a `float` to `int` removes the fractional part without rounding as shown in Listing 5.7.

Listing 5.7: Fractional part is removed by downcasting.

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

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 Listing 5.5 showed, where an integer was casted to a float while retaining its value. 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 $y.x \in [y.5, y+1)$ to $y+1$. This can be performed by downcasting as shown in Listing 5.8.

· downcasting
· upcasting
· rounding
· whole part
· fractional part

Listing 5.8: Fractional part is removed by downcasting.

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

As the example shows, for floating points whose fractional part is equal to or larger than $y.5$ adding 0.5 will make them above $(y+1).0$, and downcasting will thus downcase to $(y+1).0$. Conversely fractional pars below will downcast to $y.0$. Thus, rounding is achieved by downcasting.

5.2 Operators on basic types

Listing 5.8 is an example of an arithmetic *expression* using an *binary operator* written using *infix notation*, since the operator appears in between the *operands*. The "+" operator is binary, since it takes two arguments, and since it is written between its arguments, then it uses infix notation. Expressions is the basic building block of all F# programs and this section will discuss operator expressions on basic types.

· expression
· binary operator
· infix notation
· operands

The syntax of basic binary operators is shown in Listing 5.9.

Listing 5.9 Syntax for a binary expression.

```
<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. Not all operators are defined for all types, e.g., addition is defined for integer and float types as well as for characters and strings, but multiplication is only defined for integer and floating-point types. A complete list of built-in operators on basic types is shown in Table E.1 and E.2 and a range of mathematical functions shown in Table E.3. An example is 3+4. Note that expressions can themselves be arguments to expressions , and thus, 4+5+6 is also a legal statement. This is called *recursion*, which means that a rule or a function is used by the rule or function itself in its definition. See Chapter 13 for more on recursive functions. · recursion

Unary operators takes only one argument and have the syntax shown in Listing 5.10

Listing 5.10 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*. · prefix operator

The concept of *precedence* is an important concept in arithmetic expressions.[1] If parentheses are omitted in Listing 5.8, then F# will interpret the expression as (int 357.6) + 0.5, which is erroneous, since addition of an integer with a float is undefined. This is an example of precedence, i.e., function evaluation takes precedence over addition meaning that it is performed before addition. Consider the arithmetic expression, · precedence

Listing 5.11: A simple arithmetic expression.

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

Here, the addition and multiplication functions are shown in infix notation with the *operator* lexemes "+" and "*". To arrive at the resulting value 23, F# has to decide in which order to perform the calculation. There are 2 possible orders, 3 + (4 * 5) or (3 + 4) * 5, which gives different results. For integer arithmetic, the correct order is of course to multiply before addition, and we say that multiplication takes *precedence* over addition. Every atomic operation that F# can perform is ordered in terms of its precedence, and for some common built-in operators shown in Table 5.4, the precedence is shown by the order they are given in the table. · operator · precedence

Associativity implies the order in which calculations are performed for operators of same precedence. For some operators and type combinations association matters little, e.g., multiplication associates to the left and exponentiation associates to the right, as demonstrated in Listing 5.12.

[1] Jon: **minor comment on indexing and slice-ranges.**

<table>
<tr><th>Operator</th><th>Associativity</th><th>Description</th></tr>
<tr><td>+<expr>, -<expr>,
~~~<expr></td><td>Left</td><td>Unary identity, negation, and bitwise negation operator</td></tr>
<tr><td>f <expr></td><td>Left</td><td>Function application</td></tr>
<tr><td><expr> ** <expr></td><td>Right</td><td>Exponent</td></tr>
<tr><td><expr> * <expr>,
<expr> / <expr>,
<expr> % <expr></td><td>Left</td><td>Multiplication, division and remainder</td></tr>
<tr><td><expr> + <expr>,
<expr> - <expr></td><td>Left</td><td>Addition and subtraction binary operators</td></tr>
<tr><td><expr> ^^^ <expr></td><td>Right</td><td>bitwise exclusive or</td></tr>
<tr><td><expr> < <expr>,
<expr> <= <expr>,
<expr> > <expr>,
<expr> >= <expr>,
<expr> = <expr>,
<expr> <> <expr>,
<expr> <<< <expr>,
<expr> >>> <expr>,
<expr> &&& <expr>,
<expr> ||| <expr> ,</td><td>Left</td><td>Comparison operators, bitwise shift, and bitwise 'and' and 'or'.</td></tr>
<tr><td><expr> && <expr></td><td>Left</td><td>Boolean and</td></tr>
<tr><td><expr> || <expr></td><td>Left</td><td>Boolean or</td></tr>
</table>

Table 5.4: 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 has same precedence. Full table is given in Table E.5.

Listing 5.12: Precedence rules define implicit parentheses.

```
> 3.0*4.0*5.0;;
val it : float = 60.0

> (3.0*4.0)*5.0;;
val it : float = 60.0

> 3.0*(4.0*5.0);;
val it : float = 60.0

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

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

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

the expression for `3.0 * 4.0 * 5.0` associates to the left, and thus is interpreted as `(3.0 * 4.0) * 5.0`, but gives the same results as `3.0 * (4.0 * 5.0)`, since association does not matter for multiplication of numbers. However, the expression for `4.0 ** 3.0 ** 2.0` associates to the right, and thus is interpreted as `4.0 ** (3.0 ** 2.0)`, which is quite different from `(4.0 ** 3.0) ** 2.0`. **Whenever in doubt of association or any other basic semantic rules, it is a good idea to use parentheses as here. It is also a good idea to test your understanding of the syntax and semantic**

Advice

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

rules by making a simple script.

5.3 Boolean arithmetic

Boolean arithmetic is the basis of almost all computers and particularly important for controlling program flow, which will be discussed in Chapter 8. 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 tabular form, known as a *truth table*, and the truth tables for the basic Boolean operators and functions are shown in Table 5.5. 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 Boolean 'or' operator, e.g., true and false in this mnemonic translates to $1 \cdot 0 = 0$, and the results translates back to the Boolean value false. In F# the truth table for the basic Boolean operators can be produced by a program as shown in Listing 5.13.

· and
· or
· not
· truth table

Listing 5.13: Boolean operators and truth tables.

```
> printfn "a b a*b a+b not a"
- printfn "%A %A %A %A %A"
-   false false (false && false) (false || false) (not false)
- printfn "%A %A %A %A %A"
-   false true (false && true) (false || true) (not false)
- printfn "%A %A %A %A %A"
-   true false (true && false) (true || false) (not true)
- printfn "%A %A %A %A %A"
-   true true (true && true) (true || true) (not true);;
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
val it : unit = ()
```

Here, we used the `printfn` function, to 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 6.5 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 belongs together. This is an example of the so-called lightweight syntax. Generally, F# ignores newlines and whitespaces except when using the lightweight syntax, and the examples of the difference between regular and lightweight syntax is discussed in Chapter 6.

5.4 Integer arithmetic

The set of integers is infinitely large, but since all computers have limited resources, it is not possible to represent it in its entirety. The various integer types listed in Table 5.1 are finite subsets reduced by limiting their ranges. An in-depth description of integer implementation can be found in Appendix B. The type `int` is the most common type.

Table E.1, E.2, and E.3 give examples 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 straight forward. However, performing arithmetic operations on integers requires extra care, since the result may cause *overflow* and *underflow*. E.g., the range of the integer type `sbyte` is $[-128 \ldots 127]$, which causes problems in the example in Listing 5.14.

· overflow
· underflow

Listing 5.14: Adding integers may cause overflow.

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

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

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

Here $100 + 30 = 130$, which is larger than the biggest `sbyte`, and the result is an overflow. Similarly, we get an underflow, when the arithmetic result falls below the smallest value storable in an `sbyte` as demonstrated in Listing 5.15.

Listing 5.15: Subtracting integers may cause underflow.

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

I.e., we were expecting a negative number, but got a positive number instead.

The overflow error in Listing 5.14 can be understood in terms of the binary representation of integers: In binary, $130 = 10000010_2$, and this binary pattern is interpreted differently as `byte` and `sbyte`, see Listing 5.16.

Listing 5.16: The left most bit is interpreted differently for signed and unsigned integers, which gives rise to potential overflow errors.

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

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

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

The division and remainder operators, which discards the fractional part after division, and the *remainder* operator calculates the remainder after integer division, as demonstrated in Listing 5.17.

· integer division
· remainder

Listing 5.17: Integer division and remainder operators.

```
> 7 / 3;;
val it : int = 2

> 7 % 3;;
val it : int = 1
```

Together integer division and remainder is a lossless representation of the original number, see Listing 5.18.

Listing 5.18: Integer division and remainder is a lossless representation of an integer, compare with Listing 5.17.

```
> (7 / 3) * 3;;
val it : int = 6

> (7 / 3) * 3 + (7 % 3);;
val it : int = 7
```

And we see that integer division of 7 by 3 followed by multiplication by 3 is less that 7, and 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 with 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 Listing 5.19.

· exception

Listing 5.19: Integer division by zero causes an exception runtime error.

```
> 3/0;;
System.DivideByZeroException: Attempted to divide by zero.
  at <StartupCode$FSI_0002>.$FSI_0002.main@ () [0x00001] in
   <da48886c466e4b80be4566752c596fa8>:0
  at (wrapper managed-to-native)
   System.Reflection.MonoMethod:InternalInvoke
   (System.Reflection.MonoMethod,object,object[],System.Exception&)
  at System.Reflection.MonoMethod.Invoke (System.Object obj,
   System.Reflection.BindingFlags invokeAttr, System.Reflection.Binder
   binder, System.Object[] parameters, System.Globalization.CultureInfo
   culture) [0x00032] in <c9f8153c41de4f8cbafd0e32f9bf6b28>:0
Stopped due to error
```

The output looks daunting at first sight, but the first and last line of the error message are the most important parts, which tells us what exception was cast and why the program stopped. The middle are technical details concerning which part of the program caused this, and can be ignored for the time being. Exceptions are a type of *runtime error*, and are treated in Chapter 18

· runtime error

Integer exponentiation is not defined as an operator, but this is available the built-in function `pown`.

a	b	a ~~~ b
false	false	false
false	true	true
true	false	true
true	true	false

Table 5.6: Boolean exclusive or truth table.

This function is demonstrated in Listing 5.20.

Listing 5.20: Integer exponent function.

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

which is equal to 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> &&& <expr>`, bitwise 'and', returns the result of taking the Boolean 'and' operator position-wise; `<expr> ||| <expr>`, bitwise 'or', as 'and' but using the Boolean 'or' operator; and `<expr> ~~~ <expr>`, bitwise xor, which is returns the result of the Boolean 'xor' operator defined by the truth table in Table 5.6.

· xor
· exclusive or

5.5 Floating point arithmetic

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

Table E.1, E.2, and E.3 give examples operators and functions pre-defined for floating point types. Note that the remainder operator for floats calculates the remainder after division and discarding the fractional part, see Listing 5.21.

Listing 5.21: Floating point division and remainder operators.

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

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

The remainder for floating point numbers can be fractional, but division, discarding fractional part, and remainder is still a lossless representation of the original number as demonstrated in Listing 5.22.

Listing 5.22: Floating point division, truncation, and remainder is a lossless representation of a number.

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

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

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

Arithmetic using `float` will not cause over- and underflow problems, since the IEEE 754 standard includes the special numbers $\pm\infty$ and NaN. As shown in Listing 5.23, no exception is thrown.

Listing 5.23: Floating point numbers include infinity and Not-a-Number.

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

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

However, the `float` type has limited precision, since there is only a finite number of numbers that can be stored in a float. E.g., addition and subtraction can give surprising results as demonstrated in Listing 5.24.

Listing 5.24: Floating point arithmetic has finite precision.

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

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 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 number but not 0. Such errors tend to accumulate and comparing the result of expressions of floating point values should therefore be treated with care. Thus, **equivalence of two floating point expressions should only be considered up to sufficient precision, e.g., comparing `357.8 + 0.1` and `357.9` up to `1e-10` precision should be tested as, `abs ((357.8 + 0.1) - 357.9) < 1e-10`.** Advice

5.6 Char and string arithmetic

Addition is the only operator defined for characters. Nevertheless, character arithmetic is often done by casting to integer. A typical example is conversion of case, e.g., to convert the lowercase character 'z' to uppercase, we use the *ASCIIbetical order* and add the difference between any Basic Latin Block letters in upper- and lowercase as `integers` and cast back to `char`, see Listing 5.25. · ASCIIbetical order

Listing 5.25: Converting case by casting and integer arithmetic.

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

I.e., the code 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 Listing 5.26.

Listing 5.26: Example of string concatenation.

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

Characters and strings cannot be concatenated, which is why the above example used the string of a space `" "` instead of the space character `' '`. The characters of a string may be indexed as using the `.[]` notation. This is demonstrated in Listing 5.27. · `.[]`

Listing 5.27: String indexing using square brackets.

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

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

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

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

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

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

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 Listing 5.28.

Listing 5.28: String length attribute and string indexing.

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

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

Since index counting starts at 0, and the string length is 7, then the index of the last character is 6. The is a long list of built-in functions in `System.String` for working with strings, some of which will be discussed in Section 11.1.

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 object, classes, and methods see Chapter 20.

· dot notation
· object
· class
· method

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 the `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 onto 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 short is smaller than the larger, e.g., `"abs" < "absalon"` is true, while `"abs" < "abs"` is false. The “<=”, “>”, and “>=” operators are defined similarly.

5.7 Programming intermezzo: Hand conversion between decimal and binary numbers

Conversion of integers between decimal and binary form is a key concept in order to understand some of the basic properties of calculations on the computer. From binary to decimal is straight forward using the power-of-two algorithm, i.e., given a sequence of $n+1$ bits that represent an integer $b_n b_{n-1} \ldots b_0$, where b_n and b_0 are the most and least significant bits, then the decimal value is calculated as,

$$v = \sum_{i=0}^{n} b_i 2^i \tag{5.1}$$

For example, $10011_2 = 1 + 2 + 16 = 19$. 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 when you divide a number by two, then that is equivalent to shifting its binary representation 1 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 that the least significant bit is the remainder after integer division by two. Sequential application of this idea leads directly to the divide-by-two algorithm. E.g., if we were to convert the number 11_{10} on decimal form to binary form we would perform the following steps:

$$\begin{aligned}
11 \text{ div } 2 = 5,\ 11 \text{ rem } 2 &= 1\\
5 \text{ div } 2 = 2,\ 5 \text{ rem } 2 &= 1\\
2 \text{ div } 2 = 1,\ 2 \text{ rem } 2 &= 0\\
1 \text{ div } 2 = 0,\ 1 \text{ rem } 2 &= 1
\end{aligned}$$

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 interactive mode, we can calculate the same as shown in Listing 5.29.

Listing 5.29: Converting the number 11_{10} to binary form.

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

Thus, but 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 may be used on the whole part, while multiply may be used in a similar manner on the fractional part.

6 Values and functions

In the previous chapter, we saw how to use F# as a calculator working with literals, operators and built-in functions. To save time and make programs easier to read and debug, it is useful to bind expressions to identifiers either as new constants, functions, or operators. As an example, consider the problem,

Problem 6.1

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

$$ax^2 + bx + c = 0 \tag{6.1}$$

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

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \tag{6.2}$$

which gives the general solution for any values of the coefficients. Here, we will assume a positive discriminant, $b^2 - 4ac > 0$. In order to write a program, where the code may be reused later, we define a function `discriminant : float -> float -> float -> float` , that is, a function that takes 3 arguments, `a`, `b`, and `c`, and calculates the discriminant. Details on function definition is given in Section 6.2. Likewise, we will define functions `positiveSolution : float -> float -> float -> float` and `negativeSolution : float -> float -> float -> float`, that also takes the polynomial's coefficients as arguments and calculates the solution corresponding to choosing the postive and negative sign for $\pm$ in the equation. Our solution thus looks like Listing 6.1.

Listing 6.1 identifiersExample.fsx:
Finding roots for quadratic equations using function name binding.

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

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

```
$ fsharpc --nologo identifiersExample.fsx && mono identifiersExample.exe
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has discriminant 4.0 and solutions -1.0 and 1.0
```

Here, we have further defined names of values `a`, `b`, and `c` used as input to our functions, and the results of function application is bound to the names `d`, `xn`, and `xp`. The names of functions and values given here are examples of identifiers, and with these, we may reuse the quadratic formulas and calculated values later, while avoiding possible typing mistakes and reducing amount of code, which needs to be debugged.

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

· identifier

Identifier

- Identifiers are used as names for values, functions, types etc.
- They must start with a letter or underscore '_', but can be followed by zero or more of letters, digits, and a range of special characters except SP, LF, and CR (space, line feed, and carriage return). Letters are most unicode codepoints that produces regular letters, see Appendix C.3 for more on codepoints.
- They can also be a sequence of identifiers separated by a period.
- They cannot be keywords, see Table 6.1.

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 **Restrict identifiers to use letters from the english alphabet, numbers, period, and '_'.** However, the number of possible identifiers is enormous: the full definition referes to the Unicode general categories described in Appendix C.3, 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.

Advice

Identifiers may be used to carry information about their intended content and use, and careful selection of identifiers can aid programmers communicate thoughts about the code. Thus, identifiers are often a word or several concatenated words from the human language. 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

<table>
<tr><th>Type</th><th>Keyword</th></tr>
<tr><td>Regular</td><td>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.</td></tr>
<tr><td>Reserved</td><td>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.</td></tr>
<tr><td>Symbolic</td><td>let!, use!, do!, yield!, return!, |, ->, <-, ., :, (,), [,], [<, >], [|, |], {, }, ', #, :?>, :?, :>, .., ::, :=, ;;, ;, =, _, ?, ??, (*), <@, @>, <@@, and @@>.</td></tr>
<tr><td>Reserved symbolic</td><td>~ and `</td></tr>
</table>

Table 6.1: Table of (possibly future) *keywords* and symbolic keywords in F#.

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 readers tradition.** Finally, identifiers are often concatenations of words, as `positiveSolution` in Listing 6.1. Concatenations can be difficult to read. Without the capitalised second word, we would have had `positivesolution`. This is readable at most times, but takes longer time for humans to parse 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 few character names, and concatenation should be emphasized, e.g., by camel casing.** 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 use short. What is complex and what is simple is naturally in the eye of the beholder, but when we program, remember that a future reader of the program most likely has not had time to work the problem as long as the programmer, thus **choose identifier names as if you were to explain the meaning of a program to a knowledgable outsider.**

Advice

· lower camel case
· mixed case
· upper camel case
· pascal case

Advice

Advice

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

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

· expression

Expressions

- An Expression is a computation such as `3 * 5`.
- They can be value bindings between identifiers and expressions that evaluate to a value or a function, see Sections 6.1 and 6.2.
- They can be do bindings that produce side-effects and whose result is ignored, see Section 6.2
- They can be assignments to variables, see Section 6.1.

- They can be a sequence of expressions separated with "`;`" lexeme.
- They can be annotated with a type using the "`:`" lexeme. · `:`

Before we begin a deeper discussion on bindings, note that F# has adheres to two different syntaxes: regular and *lightweight*. In the regular syntax, newlines and whitespaces are generally ignored, while in lightweight syntax, certain keywords and lexemes may be replaced by specific use of newlines and whitespaces. Lightweight syntax is the most common, but the syntaxes may be mixed, and we will highlight the options, when relevant. · lightweight syntax

6.1 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, · value-binding

Listing 6.2 Value binding expression.

```
let <valueIdent> = <bodyExpr> [in <expr>]
```

The *`let`* keyword binds a value-identifier `<valueIdent>` with an expression `<bodyExpr>` that evaluates to a value. If the *`in`* keyword is used, then the value-identifier becomes synonymous with the evaluated value in `<expr>` only. The square bracket notation `[]` means that the enclosed is optional. Here the meaning is that for lightweight syntax, the `in` keyword is replaced with a newline, and the binding is valid in the following lines at the level of scope of the value-binding or deeper *lexically*. · `let` · `in` · lexically

The value identifier annotated with a type 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 the `<bodyExpr>` is evaluated, but the result is discarded. See Chapter 15 for more details on patterns. · `:` · `_` · wildcard

For example, letting the identifier `p` be bound to the value `2.0` and using it in an expression is done as shown in Listing 6.3.

Listing 6.3 letValue.fsx:
The identifier `p` is used in the expression following the `in` keyword.

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

```
$ fsharpc --nologo letValue.fsx && mono letValue.exe
9.0
```

F# will ignore most newlines between lexemes, i.e., the above is equivalent to writing as shown in Listing 6.4.

Listing 6.4 letValueLF.fsx:
Newlines after in make the program easier to read.

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

```
$ fsharpc --nologo letValueLF.fsx && mono letValueLF.exe
9.0
```

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 Listing 6.5.

· lightweight syntax

Listing 6.5 letValueLightWeight.fsx:
Lightweight syntax does not require the in keyword, but expression must be aligned with the let keyword.

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

```
$ fsharpc --nologo letValueLightWeight.fsx
$ mono letValueLightWeight.exe
9.0
```

The same expression in interactive mode will also respond the inferred types as, e.g., shown in Listing 6.6.

Listing 6.6: Interactive mode also responds inferred types.

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

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 of type `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. I.e., mixing types gives an error as shown in Listing 6.7.

**Listing 6.7 letValueTypeError.fsx:
Binding error due to type mismatch.**

```
let p : float = 3
do printfn "%A" (3.0 ** p)
```

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

letValueTypeError.fsx(1,17): error FS0001: This expression was expected
   to have type
    'float'
but here has type
    'int'
```

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 Listing 6.8.

**Listing 6.8 letValueSequence.fsx:
A value-binding for a sequence of expressions.**

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

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

The lightweight syntax automatically inserts the "`;`" lexeme at newlines, hence using the lightweight syntax the above is the same as shown in Listing 6.9.

**Listing 6.9 letValueSequenceLightWeight.fsx:
A value-binding for a sequence using lightweight syntax.**

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

```
$ fsharpc --nologo letValueSequenceLightWeight.fsx
$ mono letValueSequenceLightWeight.exe
2.0
9.0
```

A key concept of programming is *scope*. In F#, the scope of a value-binding is lexically meaning that when F# determines the value bound to a name, it looks left and upward in the program text for the `let` statement defining it, see, e.g., Listing 6.10. · scope

Listing 6.10 letValueScopeLower.fsx:
Redefining identifiers is allowed in lower scopes.

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

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

F# also has to option of using dynamic scope, where the value of a binding is defined by when it is used, and this will be discussed in Section 6.6.

Scopes are given levels, and scopes may be nested, where the nested scope has a level one lower than its parent.[1] F# distinguishes between the top and lower levels, and at the top level in the lightweight syntax, redefining values is not allowed, as shown in Listing 6.11.

Listing 6.11 letValueScopeLowerError.fsx:
Redefining identifiers is not allowed in lightweight syntax at top level.

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

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

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

But using parentheses, we create a *block*, i.e., a *nested scope*, and then redefining is allowed, as demonstrated in Listing 6.12.

· block
· nested scope

Listing 6.12 letValueScopeBlockAlternative3.fsx:
A block may be created using parentheses.

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

```
$ fsharpc --nologo letValueScopeBlockAlternative3.fsx
$ mono letValueScopeBlockAlternative3.exe
4
```

In both cases we used indentation, which is good practice, but not required here. Bindings inside are not available outside the nested scope as shown in Listing 6.13.

[1] Jon: **Drawings would be good to describe scope**

Listing 6.13 letValueScopeNestedScope.fsx:
Bindings inside a scope are not available outside.

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

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

letValueScopeNestedScope.fsx(6,22): error FS0039: The value or
   constructor 'q' is not defined. Maybe you want one of the following:
   p
```

Nesting is a natural part of structuring code, e.g., through function definitions to be discussed in Section 6.2 and flow control structures to be discussed in Chapter 8. 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.

Defining blocks is useful for controlling the extend of a lexical scope of bindings. For example, adding a second `printfn` statement as in Listing 6.14. 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)`.

Listing 6.14 letValueScopeBlockProblem.fsx:
Overshadowing hides the first binding.

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

```
$ fsharpc --nologo letValueScopeBlockProblem.fsx
$ mono letValueScopeBlockProblem.exe
4
4
```

Had we intended to print the two different values of `p`, the we should have create a block as in Listing 6.15.

Listing 6.15 letValueScopeBlock.fsx:
Blocks allow for the return to the previous scope.

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

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

Here, the lexical scope of `let p = 4 in ...` is for the nested scope, which ends at “)”, returning to the lexical scope of `let p = 3 in ...`.

6.2 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 6.16 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 of 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 6.17 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 parantheses, 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 Chapter 8, and higher order functions in Chapter 16.

An example of defining a function and using in interactive mode is shown in Listing 6.18.

Listing 6.18: An example of a binding of an identifier and a function.

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

and 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 sufficient for F# to be able to infer the types for the full statement. In the example, one specification is sufficient, and we could just have specified the type of the result as in Listing 6.19.

Listing 6.19 letFunctionAlterantive.fsx:
All types need most often not be specified.

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

Or even just one of the arguments as in Listing 6.20.

Listing 6.20 letFunctionAlterantive2.fsx:
Just one type is often enough for F# to infer the rest.

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

In both cases, since the + *operator* is only defined for *operands* of the same type, then when the type of either the result, any or both operands are declared, then the type of the remaining follows directly. As for values, lightweight syntax automatically inserts the keyword `in` and the lexeme “;” as shown in Listing 6.21.

· operator
· operand

Listing 6.21 letFunctionLightWeight.fsx:
Lightweight syntax for function definitions.

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

```
$ fsharpc --nologo letFunctionLightWeight.fsx
$ mono letFunctionLightWeight.exe
1221.0
```

Arguments need not always be inferred to types, but may be of generic type, which F# prefers, when *type safety* is ensured as shown in Listing 6.22.

· type safety

Listing 6.22: Type safety implies that a function will work for any type, and hence it is generic.

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

Here, the function `second` does not use the first argument `x`, which therefore can be of any type, and which F# therefore calls `'a`, and 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.

· generic function

A function may contain a sequence of expressions, but must return a value. E.g., the quadratic formula may be written as shown in Listing 6.23.

Listing 6.23 identifiersExampleAdvance.fsx:
A function may contain sequences of expressions.

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

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

```
$ fsharpc --nologo identifiersExampleAdvance.fsx
$ mono identifiersExampleAdvance.exe
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has solutions -0.7071067812 and 0.7071067812
```

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 is does not matter, but all lines following the first must use the same. The scope ends before the first line with the previous indentation or none. Notice how the last expression is not bound to an identifier, but is the result of the function, i.e., in contrast to many other languages, F# does not have an explicit keyword for returning values, but requires a final expression, which will be returned to the caller of the function. Note also that since the function `discriminant` is defined in the nested scope of `solution`, then `discriminant` cannot be called outside `solution`, since the scope ends before `let a = 1.0`.

Lexical scope and function definitions can be a cause of confusion as the following example in Listing 6.24 shows.[2]

· lexical scope

Listing 6.24 lexicalScopeNFunction.fsx:
Lexical scope means that $f(z) = 3x$ and not $4x$ at the time of calling.

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

```
$ fsharpc --nologo lexicalScopeNFunction.fsx
$ mono lexicalScopeNFunction.exe
6.0
```

[2]Jon: **Add a drawing or possibly a spell-out of lexical scope here.**

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`, it has 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.** I.e., since `a` and `3.0` are synonymous in the first lines of the program, then the function `f` is really defined as, `let f z = 3.0 * z`. Advice

Functions do not need a name, but may be declared as an *anonymous function* using the `fun` keyword and the "`->`" lexeme, as shown in Listing 6.25.

· anonymous function
· `fun`
· `->`

Listing 6.25 functionDeclarationAnonymous.fsx:
Anonymous functions are functions as values.

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

```
$ fsharpc --nologo functionDeclarationAnonymous.fsx
$ mono functionDeclarationAnonymous.exe
5
```

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 new values may be reassigned to their identifiers, when mutable, as will be discussed in Section 6.6. A common use of anonymous functions is as as arguments to other functions, as demonstrated in Listing 6.26.

Listing 6.26 functionDeclarationAnonymousAdvanced.fsx:
Anonymous functions are often used as arguments for other functions.

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

```
$ fsharpc --nologo functionDeclarationAnonymousAdvanced.fsx
$ mono functionDeclarationAnonymousAdvanced.exe
18
```

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 powerfull concepts, but tend to make programs harder to read, and their use should be limited.** Advice

The result of one function is often used as an argument of another. This is function composition, and an example is shown in Listing 6.27.

Listing 6.27 functionComposition.fsx:
Composing functions using intermediate bindings.

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

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

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

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 operator has been invented called the *piping* operators, “>/” and “</”. They are used as demonstrated in Listing 6.28.

· piping
· lstinline|>
· <|

Listing 6.28 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
```

The example shows regular composition and left-to-right and right-to-left piping. The word piping is a picture of data flowing through pipes, where functions are places, where the pipes have been assembled and the data changes. The three expressions in Listing 6.28 performes the same calculation. The left-to-right piping in line 5 corresponds to the left-to-right reading direction of many human languages, i.e., the value `2` is used as argument to `f`, and the result is used as argument to `g`. However, this is opposite arithmetic composition in line 4. Right-to-left piping has the order of arithmetic composition. Unfortunately, since F# evaluates the expression from the left, without the parenthesis in line 6, `g <| f <| 2`, then F# would read the expression as `(g <| f) <| 2`, which is an error, since `g` takes an integer as argument not a function. F# can also define composite on a function level, further discussion on this is deferred to Chapter 16. The piping operator comes in four variants: “|>|”, “<||”, “||>|”, and “<|||”. The allow for piping between pairs and triples to functions of 2 and 3 arguments, see Listing 6.29 for an example.

Listing 6.29 functionTuplePiping.fsx:
Tuples can be piped to functions of more than one argument.

```
let f x = printfn "%A" x
let g x y = printfn "%A %A" x y
let h x y z = printfn "%A %A %A" x y z

1 |> f
(1, 2) ||> g
(1, 2, 3) |||> h
```

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

The example demonstrates left-to-right piping, right-to-left works analogously.[3]

A *procedure* is a generalisation of the concept of functions, and in contrast to functions procedures need not return values. This is demonstrated in Listing 6.30.

· procedure

Listing 6.30 procedure.fsx:
A procedure is a function that has no return value, and in F# returns "()".

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

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

In F# this is automatically given the unit type as return value. Procedural thinking is useful for *encapsulation* of scripts, but is prone to *side-effects* and should be minimized by being replaced by functional thinking. More on side-effects in Section 6.6. **Procedural thinking is useful for encapsulation, but is prone to side-effects and should be minimized by being replaced by functional thinking.**

· encapsulation
· side-effects
Advice

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 *closures*. A closure is a description of the function, its arguments, its expression, and the environment, at the time it was created, i.e., the tripple: $(args, exp, env)$. Consider the listing in Listing 6.31.

· first-class citizens
· closures

[3]Jon: **Tuples have not yet been introduced!**

Listing 6.31 functionFirstClass.fsx:
The function `timesTwo` 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
```

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

$$\texttt{mul} : (\text{args}, \text{exp}, \text{env}) = \big((\text{x}, \text{y}), (\texttt{x * y}), ()\big) \tag{6.3}$$

$$\texttt{applyFactor} : (\text{args}, \text{exp}, \text{env}) = \left((\text{x}, \texttt{fct}), \begin{pmatrix}\texttt{let a = fct factor x} \\ \texttt{string a}\end{pmatrix}, (\texttt{factor} \to 2.0)\right) \tag{6.4}$$

The function `mul` does not use its environment, and everything needed to evaluate its expression are values for its arguments. The function `applyFactor` also takes two arguments, a function and a value. It uses `factor` from the environment, thus this is stored in its closure. When `mul` is given as an argument in Listing 6.31 line 8, then it is its closure, which is given to `applyFactor`, and the closure contains everything that `applyFactor` requires to use `mul`. Likewise, if `applyFactor` is given as argument to yet another function, then its closure includes the relevant part of its environment at time of definition, i.e., `factor`, such that when `applyFactor` is applied to two arguments, then its closure contains everything needed to evaluate its expression.

6.3 Operators

Operators are functions, and in F#, the infix multiplication operator `+` is equivalent to the function `(+)`, e.g.,

Listing 6.32 addOperatorNFunction.fsx:
Operators have function equivalents.

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

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

All operators have this option, and you may redefine them and define your own operators, but in F#

names of user-defined operators are limited.

· operator

operator

- A *unary operator* name can be: “+”, “-”, “+.”, “-.”, “%”, “&”, “&&”, “~~”, “~~~”, “~~~~”, . . . , apostropheOp. Here apostropheOp is an operator name starting with “!” followed by one or more of either “!”, “%”, “&”, “*”, “+”, “-”, “.”, “/”, “<”, “=”, “>”, “@”, “^”, “|”, “~”, but apostropheOp cannot be “!=”. · unary operator
- An *binary operator* name can be: “+”, “-”, “+.”, “-.”, “%”, “&”, “&&”, “:=”, “::”, “$”, “?”, dotOp. Here dotOp is an operator name starting with “.” followed by “+”, “-”, “+.”, “-.”, “%”, “&”, “&&”, “-”, “+”, “||”, “<”, “>”, “=”, “|”, “&”, “^”, “*”, “/”, “%”, “!=”. Only “?” and “?<-” may start with “?”. · binary operator

The precedence rules and associativity of user-defined operators follows the rules for which they share prefixes with built-in rules, see Table E.5. E.g., `.*`, `+++`, and `<+` are valid operator names for infix operators, they have precedence as ordered, and their associativity 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,

Listing 6.33 operatorDefinitions.fsx:
Operators may be (re)defined by their function equivalent.

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

```
$ fsharpc --nologo operatorDefinitions.fsx
$ mono operatorDefinitions.exe
13
16
true
2
```

Operators beginning with `*` must use a space in its definition, `( *` in order for it not to be confused with the beginning of a comment `(*`, see Chapter 7 for more on comments in 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.** In Chapter 20 we will discuss how to define type specific operators including prefix operators. Advice

6.4 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.

· `do`
· do binding
· statement

Listing 6.34 Syntax for do bindings.

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

The expression <expr> must return **unit**. The keyword do is optional in most cases, but using it emphasises 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.

6.5 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 - the format string,

· printf

Listing 6.35 printf statement.

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

where a formatString is a string (simple or verbatim) with placeholders. The function printf prints formatString to the console, where all placeholder has been replaced by the value of the corresponding argument formatted as specified, e.g., in **printfn** "1 2 %d" 3 the formatString is "1 2 %d", and the placeholder is %d, and the **printf** replaced the placeholder with the value of the corresponding argument, and the result is printed to the console, in this case 1 2 3. There are specifiers for all the basic types and more as elaborated in Table 6.2. The placeholder can be given a specified with, either by setting a specific integer, or using the * character, indicating that the with is given as an argument prior to the replacement value. E.g., the placeholder string %8s will print a right-aligned string that is eight characters wide padded with spaces, if needed. For floating point numbers, %8f will print a number that is exactly seven digits and a decimal making eight characters in total. Zeros are added after the decimal as needed. Alternatively, we may specify the number of decimals, such that %8.1f will print a floating point number, aligned to the right with one digit after the decimal padded with spaces where needed. 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. Examples of some of these combinations are shown in Listing 6.36.

Specifier	Type	Description
%b	bool	Replaces with boolean value
%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.
%M	decimal	
%O	Objects ToString method	
%A	any built-in types	Formatted as a literal type
%a	Printf.TextWriterFormat ->'a -> ()	
%t	(Printf.TextWriterFormat -> ()	

Table 6.2: Printf placeholder string

Listing 6.36 printfExample.fsx:
Examples of printf and some of its formatting options.

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

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

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`	Print to `stderr`
`eprintfn`		as `eprintf` but with added newline.
`sprintf`	`printf "%d apples" 3`	Return printed string
`failwithf`	`failwithf "%d failed apples" 3`	prints to a string and used for raising an exception.

Table 6.3: The family of printf functions.

Not all combinations of flags and identifier types are supported, e.g., strings cannot have number of integers after the decimal specified. The placeholder types `"%A"`, `"%a"`, and `"%t"` are special for F#, examples of their use are shown in Listing 6.37.

Listing 6.37 printfExampleAdvance.fsx:
Custom format functions may be used to specialise output.

```
let noArgument writer = printf "I will not print anything"
let customFormatter writer arg = printf "Custom formatter got: \"%A\""
   arg
printf "Print examples: %A, %A, %A\n" 3.0m 3uy "a string"
printf "Print function with no arguments: %t\n" noArgument
printf "Print function with 1 argument: %a\n" customFormatter 3.0
```

```
$ fsharpc --nologo printfExampleAdvance.fsx
$ mono printfExampleAdvance.exe
Print examples: 3.0M, 3uy, "a string"
Print function with no arguments: I will not print anything
Print function with 1 argument: Custom formatter got: "3.0"
```

The `%A` is special in that all built-in types including tuples, lists, and arrays to be discussed in Chapter 11 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 predefine 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 6.3. The function `fprintf` prints to a stream, e.g., `stderr` and `stdout`, of type `System.IO.TextWriter`. Streams will be discussed in further detail in Chapter 19. The function `failwithf` is used with exceptions, see Chapter 18 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)` · `ignore`

6.6 Variables

The *`mutable`* in `let` bindings means that the identifier may be rebound to a new value using the "`<-`" lexeme with the following syntax,[4]

· `mutable`
· `<-`

Listing 6.38 Syntax for defining variables and their initial value.

```
let mutable <ident> = <expr> [in <expr>]
```

and values are changed using the assignment operator,

Listing 6.39 Value reassignment for mutable variables.

```
<ident> <- <ident>
```

Mutable data is synonymous with the term *variable*. A variable is an area in the computers working memory associated with an identifier and a type, and this area may be read from and written to during program execution, see Listing 6.40 for an example.

· mutable data
· variable

Listing 6.40 mutableAssignReassingShort.fsx:
A variable is defined and later reassigned a new value.

```
let mutable x = 5
printfn "%d" x
x <- -3
printfn "%d" x
```

```
$ fsharpc --nologo mutableAssignReassingShort.fsx
$ mono mutableAssignReassingShort.exe
5
-3
```

Here, an area in memory was denoted `x`, initially assigned the integer value 5, hence the type was inferred to be `int`. Later, this value of `x` was replaced with another integer using the "`<-`" lexeme. The "`<-`" lexeme is used to distinguish the assignment from the comparison operator, i.e., if we by mistake had written as shown in Listing 6.41, then we instead would have obtained the default assignment of the result of the comparison of the content of `a` with the integer 3, which is false.

Listing 6.41: It is a common error to mistake "=" and "<-" lexemes for mutable variables.

```
> let mutable a = 0
- a = 3;;
val mutable a : int = 0
val it : bool = false
```

However, it is important to note, that when the variable is initially defined, then the '"=" operator must be used, while later reassignments must use the "`<-`" expression.

[4]Jon: **Discussion on heap and stack should be added here.**

Assignment type mismatches will result in an error as demonstrated in Listing 6.42.

Listing 6.42 mutableAssignReassingTypeError.fsx:
Assignment type mismatching causes a compile time error.

```
let mutable x = 5
printfn "%d" x
x <- -3.0
printfn "%d" x
```

```
$ fsharpc --nologo mutableAssignReassingTypeError.fsx

mutableAssignReassingTypeError.fsx(3,6): error FS0001: This expression was expected to have type
    'int'
but here has type
    'float'
$ mono mutableAssignReassingTypeError.exe
Cannot open assembly 'mutableAssignReassingTypeError.exe': No such file or directory.
```

I.e., once the type of an identifier has been declared or inferred, then it cannot be changed.

A typical variable is a counter of type integer, and a typical use of counters is to increment them, see Listing 6.43 for an example.

Listing 6.43 mutableAssignIncrement.fsx:
Variable increment is a common use of variables.

```
let mutable x = 5 // Declare a variable x and assign the value 5 to it
printfn "%d" x
x <- x + 1 // Assign a new value -3 to x
printfn "%d" x
```

```
$ fsharpc --nologo mutableAssignIncrement.fsx
$ mono mutableAssignIncrement.exe
5
6
```

Using variables in expressions as opposed to the left-hand-side of an assignment operation, reads the value of the variable. Thus, when using a variable as the return value of a function, then the value is copied from the local scope of the function to the scope from which it is called. This is demonstrated in Listing 6.44.

Listing 6.44: Returning a mutable variable returns its value.

```
> let g () =
-   let mutable y = 0
-   y
- printfn "%d" (g ());;
0
val g : unit -> int
val it : unit = ()
```

In the example, we see that the type is a value, and not mutable.

Variables implement dynamic scope, e.g., in comparison with the lexical scope, where the value of an identifier depends on *where* it is defined, dynamic scope depends on, *when* it is used. E.g., the script in Listing 6.24 defines a function using lexical scope and returns the number `6.0`, however, if `a` is made `mutable`, then the behavior is different as shown in Listing 6.45.

Listing 6.45 dynamicScopeNFunction.fsx:
Mutual variables implement dynamics scope rules. Compare with Listing 6.24.

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

```
$ fsharpc --nologo dynamicScopeNFunction.fsx
$ mono dynamicScopeNFunction.exe
8.0
```

Here, the respons is `8.0`, since the value of `a` changed befor the function `f` was called.

6.7 Reference cells

F# has a variation of mutable variables called *reference cells*. Reference cells have built-in function ***`ref`*** and operators "`!`" and "*`:=`*", where `ref` creates a reference variable, and the "“`!`”" and the "“`:=`”" operators reads and writes its value. An example of using reference cells is given in Listing 6.46.

· reference cells
· `ref`
· `:=`

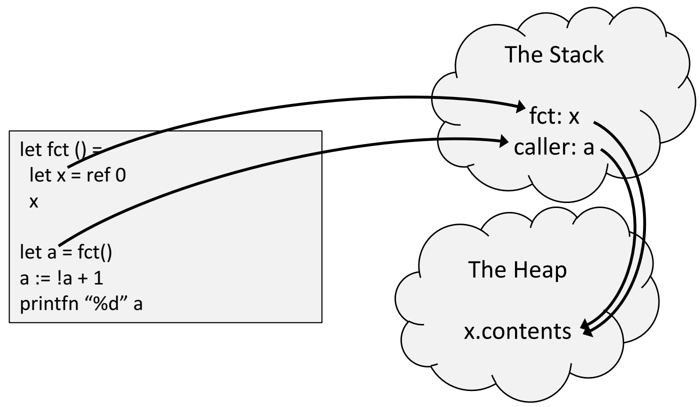


Figure 6.1: A reference cell is a pointer to The Heap and the content is not destroyed when its reference falls out of scope.

Listing 6.46 refCell.fsx:
Reference cells are variants of mutable variables.

```
let x = ref 0
printfn "%d" !x
x := !x + 1
printfn "%d" !x
```

```
$ fsharpc --nologo refCell.fsx && mono refCell.exe
0
1
```

Reference cells are different from mutable variables, since their content is allocated on *The Heap*. The Heap is a global data storage that is not destroyed, when a function returns, which is in contrast to the *call stack* also known as *The Stack*. The Stack maintains all the local data for a specific instance of a function call, see Section 13.2 for more detail on the call stack. As a consequence, when a reference cell is returned from a function, then it is the reference to the location on The Heap, which is returned as a value. Since this points outside the local data area of the function, then this location is still valid after the function returns, and the variable stored there is accessible to the caller. This is illustrated in Figure 6.1

· The Heap
· call stack
· The Stack

Reference cells may cause *side-effects*, where variable changes are performed across independent scopes. Some side-effects are useful, e.g., the `printf` family changes the content of the screen, and the screen is outside the scope of the caller. Another example of a useful side-effect is a counter shown in Listing 6.47.

· side-effects

Listing 6.47 refEncapsulation.fsx:
An increment function with a local state using a reference cell.

```
let incr =
  let counter = ref 0
  fun () ->
    counter := !counter + 1
    !counter
printfn "%d" (incr ())
printfn "%d" (incr ())
printfn "%d" (incr ())
```

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

Here `incr` is an anonymous function with an internal state `counter`. At first glance, it may be surprising, that `incr ()` does not return the value `1` at every call. The reason is that the value of the `incr` is the closure of the anonymous function `fun () -> counter := ...`, which is

$$\texttt{incr} : (\text{args}, \text{exp}, \text{env}) = \left((), \left(\begin{smallmatrix}\texttt{counter := !counter + 1}\\ \texttt{!counter}\end{smallmatrix}\right), (\texttt{counter} \rightarrow \texttt{ref 0})\right). \tag{6.5}$$

Thus, `counter` is only initiated once at the initial binding, while every call of `incr ()` updates its value on The Heap. Such a programming structure is called *encapsulation*, since the `counter` state has been encapsulated in the anonymous function, and the only way to access it is by calling the same anonymous function. **Encapsulation is good programming practice, but side-effects should be avoided wherever possible.**

· encapsulation

Advice

The `incr` example in Listing 6.47 is an example of a useful side-effect. An example to be avoided is shown in Listing 6.48.

Listing 6.48 refSideEffect.fsx:
Intertwining independent scopes is typically a bad idea.

```
let updateFactor factor =
  factor := 2

let multiplyWithFactor x =
  let a = ref 1
  updateFactor a
  !a * x

printfn "%d" (multiplyWithFactor 3)
```

```
$ fsharpc --nologo refSideEffect.fsx && mono refSideEffect.exe
6
```

In the example, the function `updateFactor` changes a variable in the scope of `multiplyWithFactor`, which is prone to errors, since the computations are not local at the place of writing, i.e., in `multiplyWithFactor`, and if `updateFactor` were defined in a library, then the source code may not be available. Better style of programming is shown in Listing 6.49.

Listing 6.49 refWithoutSideEffect.fsx:
A solution of Listing 6.48 avoiding side-effects.

```
let updateFactor () =
  2

let multiplyWithFactor x =
  let a = ref 1
  a := updateFactor ()
  !a * x

printfn "%d" (multiplyWithFactor 3)
```

```
$ fsharpc --nologo refWithoutSideEffect.fsx
$ mono refWithoutSideEffect.exe
6
```

Here, there can be no doubt in `multiplyWithFactor` that the value of `a` is changing. Side-effects do have their use, but should in general be avoided at almost all costs, and in general it is advised to **minimize the use of side effects**. Advice

Reference cells gives rise to an effect called *aliasing*, where two or more identifiers refer to the same data as illustrated in Listing 6.50. · aliasing

Listing 6.50 refCellAliasing.fsx:
Aliasing can cause surprising results and should be avoided.

```
let a = ref 1
let b = a
printfn "%d, %d" !a !b
b := 2
printfn "%d, %d" !a !b
```

```
$ fsharpc --nologo refCellAliasing.fsx && mono refCellAliasing.exe
1, 1
2, 2
```

Here, `a` is defined as a reference cell, and by defining `b` to be equal to `a`, we have created an alias. This can be very confusing, since as the example shows, changing the value of `b` causes `a` to change as well. Aliasing is a variant of side-effects, and **aliasing should be avoided at all costs**. Advice

Since F# version 4.0, the compiler has automatically converted mutable variables to reference cells, when needed. E.g., Listing 6.47 can be rewritten using a mutable variable as shown in Listing 6.51.

Listing 6.51 mutableEncapsulation.fsx:
Local mutable content can be indirectly accessed outside its scope.

```
let incr =
  let mutable counter = 0
  fun () ->
    counter <- counter + 1
    counter
printfn "%d" (incr ())
printfn "%d" (incr ())
printfn "%d" (incr ())
```

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

Reference cells is preferred over mutable variables for encapsulation to avoid confusion.

6.8 Tuples

Tuples are a direct extension of constants. They are immutable and do not have concatenations nor indexing operations. Tuples are unions of immutable types and have the following syntax, · tuple

Listing 6.52

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

Tuples are identified by the "`,`" lexeme and often enclosed in parentheses, but that is not required. An example is the tripel, also known as a 3-tuple, `(2,true,"hello")`. In interactive mode, the type of tuples are shown as demonstrated in Listing 6.53.

Listing 6.53: Tuple types are products of sets.

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

The values `2`, `true`, and `"hello"` are *members*, and the number of elements of a tuple is its *length*. From the response of F#, we see that the tuple is inferred to have the type `int * bool * string`. The "`*`" denotes the cartesian product between sets. Tuples can be products of any types and have lexical scope 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, where an identifier is bound to a tuple as shown in Listing 6.54. · member · length

Listing 6.54: Definition of a tuple.

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

In this a function is defined that takes 1 argument, a 3-tuple. If we wanted a function of 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 Chapter 15. Since we used the `\%A` placeholder in the `printfn` function, then 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 Section 14.6 for more on variable types.

Pairs or 2-tuples are so common that F# includes two built-in functions *`fst`* and *`snd`*, to extract the first and second element of a pair. This is demonstrated in Listing 6.55. · fst · snd

Listing 6.55 pair.fsx:
Deconstruction of pairs with the built-in functions `fst` and `snd`.

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

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

Tuples comparison are defined similarly as strings. Tuples of different lengths are different. For tuples of equal length, then they 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 the left operand should come before the right, when sorting alphabetically, see Listing 6.56 for an example.

Listing 6.56 tupleCompare.fsx:
Tuples are compared as strings are compared alphabetically.

```
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
```

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, in which case the result of `(a1, a2) < (b1, b2)` is equal to the result of `a1 < b1`. If `la1` and `b1` are equal, then we move onto the next letter and repeat the investigation. The "<=", ">", and ">=" operators are defined similarly.

Binding tuples to mutables does not make the tuple mutable. This is demonstrated in Listing 6.57.

Listing 6.57 tupleOfMutables.fsx:
A mutable change value, but the tuple defined by it does not refer to the new value.

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

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

However, it is possible to define a mutable variable of type tuple such that new tuple values can be assigned to it as shown in Listing 6.58.

**Listing 6.58 mutableTuple.fsx:
A mutable tuple can be assigned a new value.**

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

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

Mutable tuples are value types meaning that binding to new names are copies not aliases as demonstrated in Listing 6.59.

**Listing 6.59 mutableTupleValue.fsx:
A mutable tuple is a value type.**

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

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

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

· obfuscation

Advice

7 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 between 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 are either block comments,

Listing 7.1 Block comments.

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

The comment text `(*<any text>*)` can be any text and it 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 be given as line comments,

Listing 7.2 Line comments.

```
//<any text>newline
```

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 serves as comments for the code construct, that follows immediately after. XML consists of tags which always appears in pairs, e.g., the tag "tag" would look

· Extensible Markup Language
· XML

[1] For specification of C# documentations comments see ECMA-334 3rd Edition, Annex E, Section 2: `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf`

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

Table 7.1: Recommended XML tags for documentation comments, from ECMA-334 3rd Edition, Annex E, Section 2.

like `<tag> ... </tag>`. The F# accept any tags, but recommends those listed in Table 7.1. If no tags are used, then it is automatically assumed to be a `<summary>`. An example of a documented script is,

**Listing 7.3 commentExample.fsx:
Code with XML comments.**

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

/// <summary>Find x when 0 = ax^2+bx+c.</summary>
/// <remarks>Negative discriminant 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> to the console.
/// </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
```

```
$ fsharpc --nologo commentExample.fsx && mono commentExample.exe
0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 1.0
```

Mono's `fsharpc` command may be used to extract the comments into an XML file as demonstrated in Listing 7.4.

Listing 7.4, Converting in-code comments to XML.

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

This results in an XML file with the following content as shown in Listing 7.5.

Listing 7.5, An XML file generated by `fsharpc`.

```
<?xml version="1.0" encoding="utf-8"?>
<doc>
<assembly><name>commentExample</name></assembly>
<members>
<member name="M:CommentExample.solution(System.Double,System.Double,System.Double,System.Double)">
 <summary>Find x when 0 = ax^2+bx+c.</summary>
 <remarks>Negative discriminant 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> to the console.
 </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>
</member>
<member name="M:CommentExample.discriminant(System.Double,System.Double,System.Double)">
<summary>
 The discriminant of a quadratic equation with parameters a, b, and c
</summary>
</member>
</members>
</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 `<?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 Chapters 9 and 20, and `<assembly>`, `<members>`, and `<member>` are indications for where the functions belong in the hierarchy. As an example, the prefix `M:CommentExample.` means that it is a method in the namespace `commentExample`, which in this case is the name of the file. Further, 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 analyse compiled code and generate an empty XML structure with placeholders to describe functions, values, and variables. This structure can then be updated and edited as the program develops. The edited XML files can then be exported to *Hyper*

Text Markup Language (*HTML*) files, which can be viewed in any browser. Using the console, all of this is accomplished by the procedure shown in Listing 7.6, and the result is shown in Figure 7.1.

· Hyper Text Markup Language
· HTML

Listing 7.6, Converting an XML file to HTML.

```
$ mdoc update -o commentExample -i commentExample.xml commentExample.exe
New Type: CommentExample
Member Added: public static double determinant (double a, double b,
   double c);
Member Added: public static double solution (double a, double b, double
   c, double sgn);
Member Added: public static double a { get; }
Member Added: public static double b { get; }
Member Added: public static double c { get; }
Member Added: public static double xp { get; }
Namespace Directory Created:
New Namespace File:
Members Added: 6, Members Deleted: 0
$ mdoc export-html -out commentExampleHTML commentExample
.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 discriminant 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 7.1: Part of the HTML documentation as produce by `mdoc` and viewed in a browser.

8 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 the 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 Chapter 13.

8.1 While and for loops

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

Listing 8.1 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. As an example, the program in Listing 8.5 counts from 1 to 10. · condition · `do` · `done`

Listing 8.2 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
```

We will call `i` for the counter variable. The counting is done by performing the following computation: In line 1, the counter variable is first given an initial value 1. Then execution enters the while-loop and examines the condition. Since 1 <= 10 then the condition is true, and execution enters the body of the loop. The body prints the value of the counter to screen and increases 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 reach value 11, in which case the condition 11 <= 10 is false, and execution continues in line 2.

In lightweight syntax this would be as shown in Listing 8.3.

Listing 8.3 countWhileLightweight.fsx:
Count to 10 with a counter variable using lightweight syntax, see Listing 8.2.

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

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

Notice that although the 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`*-loops. For-loops comes in several variants, and here we will focus on the one using an explicit counter. Its syntax is, · `for`

Listing 8.4 For loop.

```
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, then the counter is increased and execution evaluates once again `<bodyExpr>`. This is repeated until and including the counter has the value `<lastExpr>`. As for while-loops, using lightweight syntax the block following the *`do`* keyword up to and including the *`done`* keyword may be replaced by a newline and indentation. · `do` · `done`

The counting example from Listing 8.2 using a `for`-loop is shown in Listing 8.5

Listing 8.5 count.fsx:
Counting from 1 to 10 using a `for`-loop.

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

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

As this interactive 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 8.6.

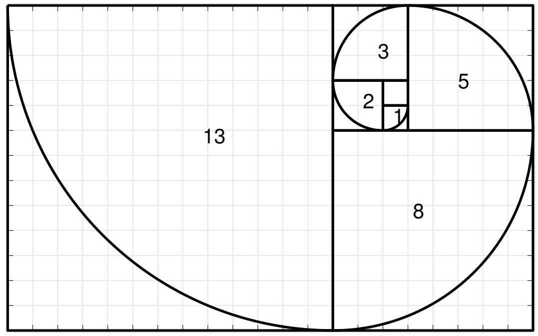


Figure 8.1: The Fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive Fibonacci numbers, and the curve in each square is the circular arc with radius of the square it is drawn in.

Listing 8.6 countLightweight.fsx:
Counting from 1 to 10 using a for-loop, see Listing 8.5.

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

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

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

Problem 8.1

Write a program that calculates the n'th Fibonacci number.

The Fibonacci's 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's numbers are related to Golden spirals shown in Figure 8.1. Often the sequence is extended with a preceding number 0, to be $0, 1, 1, 2, 3, \ldots,$ which we will do here as well.

We could solve this problem with a `for`-loop as follows,

Listing 8.7 fibFor.fsx:
The n'th Fibonacci number is the sum of the previous 2.

```
let fib n =
  let mutable pair = (0, 1)
  for i = 2 to n do
    pair <- (snd pair, (fst pair) + (snd pair))
  snd pair

printfn "fib(1) = %d" (fib 1)
printfn "fib(2) = %d" (fib 2)
printfn "fib(3) = %d" (fib 3)
printfn "fib(10) = %d" (fib 10)
```

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

The basic idea of the solution is that if we are given the $(n-1)$'th and $(n-2)$'th numbers, then the n'th number is trivial to compute. And assume that fib(1) and fib(2) are given, then it is trivial to calculate the fib(3). For the fib(4) we only need fib(3) and fib(2), hence we may disregard fib(1). Thus, we realize that we can cyclicly update the previous, current, and next values by shifting values until we have reached the desired fib(n). This is implement in Listing 8.7 as 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, `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 $n < 2$ then 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.

The same program but using a while-loop is shown in Listing 8.8.

Listing 8.8 fibWhile.fsx:
Search for the largest Fibonacci number less than a specified number.

```
1 let fib (n : int) : int =
2   let mutable pair = (0, 1)
3   let mutable i = 1
4   while i < n do
5     pair <- (snd pair, fst pair + snd pair)
6     i <- i + 1
7   snd pair
8
9 printfn "fib(1) = %d" (fib 1)
10 printfn "fib(2) = %d" (fib 2)
11 printfn "fib(3) = %d" (fib 3)
12 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
```

As can be seen, the program is almost identical. In this case, the `for`-loop is to be preferred, since more lines of code typically means more chances of making a mistake. However, while-loops are still simpler and possibly easier to argue correctness about. To understand what is being calculated in code such as the while-loop in Listing 8.8, we can describe the loop in terms of its *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 4 in Listing 8.8, 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:

· loop invariant
· invariant

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, then the body first assigns a new value to `pair` as the i'th and $i+1$'th Fibonacci numbers, then increases i with one such that at the end of the loop the `pair` again contains the the $i-1$'th and i'th Fibonacci numbers. Thus, the invariant is true.

Thus when know that the second value in `pair` holds the value of the i'th Fibonacci number, and since we further may prove that when line 7 is only reached, then $i = n$, and thus that `fib` returns the n'th Fibonacci number.

While-loops also allows for other logical structures than for-loops, such as the case when the number of iteration cannot easily be decided, when entering the loop. As an example, consider slight variation of the above problem, where we wish to find the largest Fibonacci number less than some number. An solution to this problem is shown in Listing 8.9.

Listing 8.9 fibWhileLargest.fsx:
Search for the largest Fibonacci number less than a specified number.

```
let largestFibLeq n =
  let mutable pair = (0, 1)
  while snd pair <= n do
    pair <- (snd pair, fst pair + snd pair)
  snd pair

for i = 1 to 10 do
  printfn "largestFibLeq(%d) = %d" i (largestFibLeq i)
```

```
$ fsharpc --nologo fibWhileLargest.fsx && mono fibWhileLargest.exe
largestFibLeq(1) = 2
largestFibLeq(2) = 3
largestFibLeq(3) = 5
largestFibLeq(4) = 5
largestFibLeq(5) = 8
largestFibLeq(6) = 8
largestFibLeq(7) = 8
largestFibLeq(8) = 13
largestFibLeq(9) = 13
largestFibLeq(10) = 13
```

The strategy here is to iteratively calculate numbers in Fibonacci's sequence until we've found one larger than the argument `n`, and then return the previous. This could not be calculated with a for-loop.

8.2 Conditional expressions

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

· `if`
· `then`
· `elif`
· `else`

Listing 8.10 Conditional expressions.

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

The condition `<con>` is an expression resulting in a Boolean value, and there can be zero or more `elif` conditions as indicated by `{}`. Each expression `<expr>` is called a *branch*, and all branches must have identical type, such that regardless which branch is chosen, then the type of the result of the conditional expression is the same. The result of the conditional expression is the first branch, for which its condition was true and if all conditions are false then the `else`-branch is evaluated. If no `else` expression is present, then "`()`" will be returned. See Listing 8.11 for a simple example.

· branch

Listing 8.11 condition.fsx:
Conditions evaluates their branches depending on the value of the condition.

```
if true then printfn "hi" else printfn "bye"
if false then printfn "hi" else printfn "bye"
```

```
$ fsharpc --nologo condition.fsx && mono condition.exe
hi
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 ns. This is done in Listing 8.12 using the lightweight syntax.

Listing 8.12 conditionalLightweight.fsx:
Using conditional expression to generate different strings.

```
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"
```

The sentence structure and its variants gives rise to a more compact solution, since the language to be returned to the user is a variant of "I have/or no/number apple(s)", i.e., under certain conditions should the sentence use “have” and “owe” etc. 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:

Listing 8.13 conditionalLightweightAlt.fsx:
Using sentence parts to construct the final sentence.

```
let applesIHave n =
  let haveOrOwe = if n < 0 then "owe" else "have"
  let pluralS = if (n = 0) || (abs n) > 1 then "s" else ""
  let number = if n = 0 then "no" else (string (abs n))

  "I " + haveOrOwe + " " + number + " apple" + pluralS

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 conditionalLightWeightAlt.fsx
$ mono conditionalLightWeightAlt.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"
```

While arguably shorter, this solution is also denser, and for a small problem like this, it is most likely more difficult to debug and maintain.

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` will be 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# always to include an `else` branch.

8.3 Programming intermezzo: Automatic conversion of decimal to binary numbers

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

Problem 8.2

Given an integer on decimal form, write its equivalent value on 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, by integer division by 2 and checking the remainder, we may sequentially read off the least significant bit. This leads to the algorithm shown in Listing 8.14.

Listing 8.14 dec2bin.fsx:
Using integer division and remainder to write any positive integer on 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))
```

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

In the code, the states `v` and `str` are iteratively updated until `str` finally contains the desired solution.

To prove that Listing 8.14 calculates the correct sequence we use induction. First we realize, that for $v < 1$ then the `while` loop is skipped, and the result is trivially true. We will concentrate on line 9 in Listing 8.14, and we 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 $n = 000\dots000x$: 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, `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, then `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 $n > 2^m$ then `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.

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

9 Organising code in libraries and application programs

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

· library

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 Chapter 20.

9.1 Modules

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

· module

Every implementation and script file in F# implicitly defines a module, and the module name is given by the filename. Thus, creating a script file `Meta.fsx` as shown in Listing 9.1.[1]

Listing 9.1 Meta.fsx:
A script file defining the `apply` function.

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

we've implicitly defined a module of name `Meta`. Another script file 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 shown in Listing 9.3.

Listing 9.2 MetaApp.fsx:
Defining a script calling the module.

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

In the above, we have explicitly used the module's type definition for illustration purposes. A shorter

[1]Jon: **Type definitions have not been introduced at this point!**

and possibly simpler program would have been to define `add` as `let add x y = x + y`, since F#'s typesystem will infer the implied type. However, **explicit definitions of types is recommended for readability.** Hence, an alternative to the above's use of lambda functions is, `let add (x: float) (y: float) : float = x + y`. To compile the module and the application program, we write as demonstrated in Listing 9.3.

Advice

Listing 9.3: Compiling both the module and the application code. Note that fileorder matters, when compiling several files.

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

Notice, since the 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, then 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`* using the syntax,

· `module`

Listing 9.4 Outer module.

```
module <ident>
<script>
```

Here, the identifier `<ident>` is a name not necessarily related to the filename, and the script `<script>` is expression. An example is given in Listing 9.20.

Listing 9.5 MetaExplicit.fsx:
Explicit definition of the outermost module.

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

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 Listing 9.6.

Listing 9.6: Changing the module definition to explicit naming has no effect on the application nor the compile command.

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

Notice that, 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.** I.e., 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 convention. Thus, direct linking of filenames with the internal workings of a program is a needless complication of structure.

Advice

The definitions inside a module may be accessed directly from an application program, omitting the

"."-notation, by use of the *open* keyword, · open

Listing 9.7 Open module.

```
open <ident>
```

I.e., we can modify `MetaApp.fsx` as shown in Listing 9.9.

Listing 9.8 MetaAppWOpen.fsx:
Avoiding the "."-notation by the open keyword.

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

In this case, the namespace of our 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 Listing 9.9.

Listing 9.9: How the application program opens the module has no effect on the module code nor compile command.

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

The `open`-keyword should be used sparingly, since including a library's definitions into the application scope can cause surprising naming conflicts, since 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 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**. Notice that for historical reasons, the work namespace pollution is used to cover both pollution due to modules and namespaces.

· namespace pollution
Advice

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

Listing 9.10 Nested modules.

```
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 Listing 9.11.

Listing 9.11 nestedModules.fsx:
Modules may be nested.

```
module Utilities
let PI = 3.1415
module Meta =
  type floatFunction = float -> float -> float
  let apply (f : floatFunction) (x : float) (y : float) : float = f x y
module MathFcts =
  let add : Meta.floatFunction = fun x y -> x + y
```

In this case, `Meta` and `MathFcts` 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 Listing 9.12 shows.

Listing 9.12 nestedModulesApp.fsx:
Applications using nested modules require additional usage of the "." notation to navigate the nesting tree.

```

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

Modules can be recursive using the *`rec`*-keyword, meaning that in our example we can make the outer module recursive as demonstrated in Listing 9.13.[2] · `rec`

Listing 9.13 nestedRecModules.fsx:
Mutual dependence on nested modules requires the `rec` keyword in the module definition.

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

The consequence is that the modules `Meta` and `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.** Advice

9.2 Namespaces

An alternative to structure code in modules is use a *namespace*, which only can hold modules and type · namespace

[2]Jon: **Dependence on version 4.1 and higher.**

declarations and only works in compiled mode. Namespaces are defined as explicitly defined outer modules using the *namespace* keyword,

· namespace

Listing 9.14 Namespace.

```
namespace <ident>
<script>
```

An example is given in Listing 9.15.

Listing 9.15 namespace.fsx:
Defining a namespace is similar to explicitly named modules.

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

Notice that when organising code in a namespace, the 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 Listing 9.17.

Listing 9.16 namespaceApp.fsx:
The "`.`"-notation lets the application program accessing functions and types in a namespace.

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

Likewise, compilation is performed identically, see Listing 9.17.

Listing 9.17: Compilation of files including namespace definitions uses the same procedure as modules.

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

Hence, from an application point of view, it is not immediately 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 Listing 9.18.

Listing 9.18 namespaceExtension.fsx:
Namespaces may span several files. Here is shown an extra file, which extends the `Utilities` namespace.

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

To compile we now need to include all three files in the right order. Likewise, compilation is performed identically, see Listing 9.19.

Listing 9.19: Compilation of namespaces defined in several files requires careful consideration of order, since the compiler reads once and only once through the files in the order they are given.

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

The order matters since `namespaceExtension.fsx` relies on the definition of `floatFunction` in the file `namespace.fsx`. You can use extensions to extend existing namespaces included with the F# compiler.[3][4]

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

9.3 Compiled libraries

Libraries may be distributed in compiled form as `.dll` files. This saves the user for 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 `MetaExplicitModuleDefinition.fsx` and then compile an application program, we first use the compiler's `-a` option to produce the `.dll`. A demonstration is given in Listing 9.20.

Listing 9.20: A stand-alone `.dll` file is created and used with special compile commands.

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

This produces the file `MetaExplicit.dll`, which may be linked to an application using the `-r` option during compilation, see Listing 9.21.[5]

[3] Jon: **Something about intrinsic and optional extension** `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/type-extensions`**.**

[4] Jon: **Perhaps something about the global namespace** `global`**.**

[5] Jon: **This is the MacOS option standard, Windows is slightly different.**

Listing 9.21: The library is linked to an application during compilation to produce runnable code.

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

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) using the *`#r` directive* as illustrated in Listing 9.23.

· `#r` directive

Listing 9.22 MetaHashApp.fsx:
The `.dll` file may be loaded dynamically in `.fsx` script files and in interactive mode. Nevertheless, this usage is not recommended.

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

We may now omit the explicit mentioning of the library when compiling as shown in Listing 9.23.

Listing 9.23: When using the `#r` directive, then the `.dll` file need not be explicitly included in the list of files to be compiled.

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

The `#r` directive is also used to include a library in interactive mode. However, for 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.**

Advice

In the above, 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.

· script files
· implementation files

Both script and implementation files may be augmented with *signature files*. A signature file contains no implementation but only type definitions. Signature files offers three distinct features:

· signature files

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 focusses 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 is available relating to classes, and will be discussed in Chapter 20.

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 Listing 9.28.

Listing 9.24 MetaWAdd.fs:
An implementation file including the `add` function.

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

A signature file may be automatically generated as shown in Listing 9.25.

Listing 9.25: Automatic generation of a signature file at compile time.

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

MetaWAdd.fs(4,48): warning FS0988: Main module of program is empty:
   nothing will happen when it is run
```

The warning can safely be ignored, since it is at this point not our intention to produce runnable code. The above has generated the signature file in Listing 9.28.

Listing 9.26 MetaWAdd.fsi:
An automatically generated signature file from `MetaWAdd.fs`.

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

We can generate a library using the automatically generated signature file using `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 Listing 9.30 and recompiling the `.dll` as Listing 9.28 generates no error.

Listing 9.27 MetaWAddRemoved.fsi:
Removing the type defintion for `add` from `MetaWAdd.fsi`.

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

Listing 9.28: Automatic generation of a signature file at compile time.

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

But, when using the newly created `MetaWAdd.dll` with a modified version of Listing 9.3, which does not itself supply a definition of `add` as shown in Listing 9.29, we get a syntax error, since `add` now is inaccessible to the application program. This is demonstrated in Listing 9.30.

Listing 9.29 MetaWOAddApp.fsx:
A version of Listing 9.3 without a definition of `add`.

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

Listing 9.30: Automatic generation of a signature file at compile time.

```
$ fsharpc --nologo -r MetaWAdd.dll MetaWOAddApp.fsx

MetaWOAddApp.fsx(1,25): error FS0039: The value or constructor 'add' is
   not defined.
```

10 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], but made popular in computer science by Grace Hopper, who found a moth interfering with the electronic circuits of the Harward Mark II electromechanical computer and coined the term *bug* for errors in computer programs. The original bug is shown in Figure 10.1. Software is everywhere, and errors therein have huge economic impact on our society and can threaten lives[2].

· bug

The ISO/IEC organizations have developed standards for software testing[3]. To illustrate basic concepts of software quality, consider a hypothetical route planning system. Essential factors of its quality are,

· functionality

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 finde a suitable route from point a to b?

· reliability

Reliability: Does the software work reliably over time? E.g., does the route planning software work in case of internet dropouts?

· usability

Usability: Is the software easy and intuitive to use by humans? E.g., is it easy to enter adresses and alternative routes in the software's interface?

· efficiency

[1] `https://en.wikipedia.org/wiki/Software_bug`, possibly `http://edison.rutgers.edu/NamesSearch/DocImage.php3?DocId=LB003487`

[2] `https://en.wikipedia.org/wiki/List_of_software_bugs`

[3] ISO/IEC 9126, International standard for the evaluation of software quality, December 19, 1991, later replaced by ISO/IEC 25010:2011

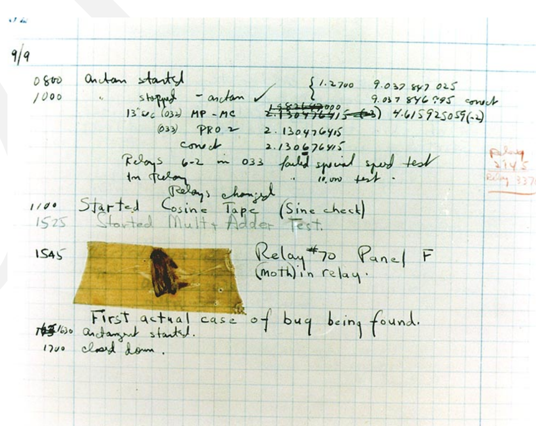


Figure 10.1: The first computer bug caught by Grace Hopper, U.S. Naval Historical Center Online Library Photograph NH 96566-KN.

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

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

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 ares 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 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.

· software testing
· debugging

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 chance of introducing new bugs. It is not exceptional that the software testing the software is as large as the software itself.

· white-box testing
· black-box testing
· unit testing

In this chapter, we will focus on two approaches to software testing, which emphasizes 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 tests 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 along side the software development.

To illustrate software testing we'll start with a problem:

Problem 10.1

Given any date in the Gregorian calendar, calculate the day of 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 knucle 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 multiplum of 4, except if it is also a multiplum of 100 but not of 400.

Many solutions to the problem have been discovered, and here we will base our program on Gauss' method, which is based on integer division and calculates the weekday of the 1st of January of a given year. For any other date, we will count our way through the weeks from the previous 1st of January. The algorithm relies on an enumeration of weekdays starting with Sunday = 0, Monday = 1, . . . , and Saturday = 6. Our proposed solution is shown in Listing 10.1.[4]

Listing 10.1 date2Day.fsx:
A function that can calculate day-of-week from any date in the Gregorian calendar.

```
let januaryFirstDay (y : int) =
  let a = (y - 1) % 4
  let b = (y - 1) % 100
  let c = (y - 1) % 400
  (1 + 5 * a + 4 * b + 6 * c) % 7

let rec sum (lst : int list) j =
  if 0 <= j && j < lst.Length then
    lst.[0] + sum lst.[1..] (j - 1)
  else
    0

let date2Day d m y =
  let dayPrefix =
    ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri"; "Satur"]
  let feb = if (y % 4 = 0) && ((y % 100 <> 0) || (y % 400 = 0)) then 29
   else 28
  let daysInMonth = [31; feb; 31; 30; 31; 30; 31; 31; 30; 31; 30; 31]
  let dayOne = januaryFirstDay y
  let daysSince = (sum daysInMonth (m - 2)) + d - 1
  let weekday = (dayOne + daysSince) % 7;
  dayPrefix.[weekday] + "day"
```

10.1 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 *coverage*. Since our program is small, we do have the opportunity to ensure that all functions are called at least once, which is called *function coverage*, we will also be able to test every branching in the program, which is called *branching coverage*, an in this case that implies *statement coverage*. The procedure is as follows:

· white-box testing
· coverage
· function coverage
· branching coverage
· statement coverage

1. Decide which are the units to test: The program shown in Listing 10.1 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 two later is superfluous. However, we may have to do this anyway, when debugging, and we may choose at

[4] Jon: **This example relies on lists, which has not been introduced yet.**

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 following code, the branch points have been given a comment and a number, as shown in Listing 10.2.

Listing 10.2 date2DayAnnotated.fsx:
In white-box testing, the branch points are identified.

```
// Unit: januaryFirstDay
let januaryFirstDay (y : int) =
  let a = (y - 1) % 4
  let b = (y - 1) % 100
  let c = (y - 1) % 400
  (1 + 5 * a + 4 * b + 6 * c) % 7

// Unit: sum
let rec sum (lst : int list) j =
  (* WB: 1 *)
  if 0 <= j && j < lst.Length then
    lst.[0] + sum lst.[1..] (j - 1)
  else
    0

// Unit: date2Day
let date2Day d m y =
  let dayPrefix =
    ["Sun"; "Mon"; "Tues"; "Wednes"; "Thurs"; "Fri"; "Satur"]
  (* WB: 1 *)
  let feb = if (y % 4 = 0) && ((y % 100 <> 0) || (y % 400 = 0)) then
     29 else 28
  let daysInMonth = [31; feb; 31; 30; 31; 30; 31; 31; 30; 31; 30; 31]
  let dayOne = januaryFirstDay y
  let daysSince = (sum daysInMonth (m - 2)) + d - 1
  let weekday = (dayOne + daysSince) % 7;
  dayPrefix.[weekday] + "day"
```

3. For each unit, produce an input set that tests each branches: In our example the branch points depends on a Boolean expression, and for good measure, we are going to test each term that can lead to branching. Thus,

<table>
<tr><th>Unit</th><th>Branch</th><th>Condition</th><th>Input</th><th>Expected output</th></tr>
<tr><td><code>januaryFirstDay</code></td><td>0</td><td><code>-</code></td><td><code>2016</code></td><td><code>5</code></td></tr>
<tr><td><code>sum</code></td><td>1</td><td><code>0 <= j && j < lst.Length</code></td><td></td><td></td></tr>
<tr><td></td><td>1a</td><td><code>true && true</code></td><td><code>[1; 2; 3] 1</code></td><td><code>3</code></td></tr>
<tr><td></td><td>1b</td><td><code>false && true</code></td><td><code>[1; 2; 3] -1</code></td><td><code>0</code></td></tr>
<tr><td></td><td>1c</td><td><code>true && false</code></td><td><code>[1; 2; 3] 10</code></td><td><code>0</code></td></tr>
<tr><td></td><td>1d</td><td><code>false && false</code></td><td><code>-</code></td><td><code>-</code></td></tr>
<tr><td><code>date2Day</code></td><td>1</td><td><code>(y % 4 = 0)</code></td><td></td><td></td></tr>
<tr><td></td><td></td><td><code>&& ((y % 100 <> 0)</code></td><td></td><td></td></tr>
<tr><td></td><td></td><td><code>|| (y % 400 = 0))</code></td><td></td><td></td></tr>
<tr><td></td><td>-</td><td><code>true && (true || true)</code></td><td><code>-</code></td><td><code>-</code></td></tr>
<tr><td></td><td>1a</td><td><code>true && (true || false)</code></td><td><code>8 9 2016</code></td><td><code>Thursday</code></td></tr>
<tr><td></td><td>1b</td><td><code>true && (false || true)</code></td><td><code>8 9 2000</code></td><td><code>Friday</code></td></tr>
<tr><td></td><td>1c</td><td><code>true && (false || false)</code></td><td><code>8 9 2100</code></td><td><code>Wednesday</code></td></tr>
<tr><td></td><td>-</td><td><code>false && (true || true)</code></td><td><code>-</code></td><td><code>-</code></td></tr>
<tr><td></td><td>1d</td><td><code>false && (true || false)</code></td><td><code>8 9 2015</code></td><td><code>Tuesday</code></td></tr>
<tr><td></td><td>-</td><td><code>false && (false || true)</code></td><td><code>-</code></td><td><code>-</code></td></tr>
<tr><td></td><td>-</td><td><code>false && (false || false)</code></td><td><code>-</code></td><td><code>-</code></td></tr>
</table>

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 test all these cases and checks the output, e.g.,

Listing 10.3 date2DayWhiteTest.fsx:
The tests identified by white-box analysis. The program from Listing 10.2 has been omitted for brevity.

```
printfn "White-box testing of date2Day.fsx"
printfn "  Unit: januaryFirstDay"
printfn "    Branch: 0 - %b" (januaryFirstDay 2016 = 5)

printfn "  Unit: sum"
printfn "    Branch: 1a - %b" (sum [1; 2; 3] 1 = 3)
printfn "    Branch: 1b - %b" (sum [1; 2; 3] -1 = 0)
printfn "    Branch: 1c - %b" (sum [1; 2; 3] 10 = 0)

printfn "  Unit: date2Day"
printfn "    Branch: 1a - %b" (date2Day 8 9 2016 = "Thursday")
printfn "    Branch: 1b - %b" (date2Day 8 9 2000 = "Friday")
printfn "    Branch: 1c - %b" (date2Day 8 9 2100 = "Wednesday")
printfn "    Branch: 1d - %b" (date2Day 8 9 2015 = "Tuesday")
```

```
$ fsharpc --nologo date2DayWhiteTest.fsx && mono
   date2DayWhiteTest.exe
White-box testing of date2Day.fsx
  Unit: januaryFirstDay
    Branch: 0 - true
  Unit: sum
    Branch: 1a - true
    Branch: 1b - true
    Branch: 1c - true
  Unit: date2Day
    Branch: 1a - true
    Branch: 1b - true
    Branch: 1c - true
    Branch: 1d - true
```

Notice that the output of the tests is organized such that they are enumerated per unit, hence we can rearrange as we like and still uniquely refer to a unit's test. Also, the output of the test program produces a list of tests that should return true or success or a similar positively loaded word, but without further or only little detail, such that we at a glance can identify any test that produced unexpected results.

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.

10.2 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:

Decide on the interface to use: It is useful to have an agreement with the software developers about what interface is to be used, e.g., in our case, the software developer has made a function `date2Day d m y`, where `d`, `m`, and `y` are integers specifying the day, month, and year.

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, 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-multiplum-of-100 leap year and in a non-leap year, a multiplum-of-100-but-not-of-400 non-leap year, and a multiplum-of-400 leap year.

Given no information about the program's text, there are other dates that one could consider as likely candidates of errors, but the above is judged to be a fair coverage.

Choose a specific set of input and expected output relations on tabular form:

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

Write a program executing the tests as shown in Listing 10.4 and 10.5.

Listing 10.4 date2DayBlackTest.fsx:
The tests identified by black-box analysis. The program from Listing 10.2 has been omitted for brevity.

```
let testCases = [
  ("A complete week",
   [(1, 1, 2016, "Friday");
    (2, 1, 2016, "Saturday");
    (3, 1, 2016, "Sunday");
    (4, 1, 2016, "Monday");
    (5, 1, 2016, "Tuesday");
    (6, 1, 2016, "Wednesday");
    (7, 1, 2016, "Thursday");]);
  ("Across boundaries",
   [(31, 12, 2014, "Wednesday");
    (1, 1, 2015, "Thursday");
    (30, 9, 2017, "Saturday");
    (1, 10, 2017, "Sunday")]);
  ("Across feburary boundary",
   [(28, 2, 2016, "Sunday");
    (29, 2, 2016, "Monday");
    (1, 3, 2016, "Tuesday");
    (28, 2, 2017, "Tuesday");
    (1, 3, 2017, "Wednesday")]);
  ("Leap years",
   [(1, 3, 2015, "Sunday");
    (1, 3, 2012, "Thursday");
    (1, 3, 2000, "Wednesday");
    (1, 3, 2100, "Monday")]);
  ]

printfn "Black-box testing of date2Day.fsx"
for i = 0 to testCases.Length - 1 do
  let (setName, testSet) = testCases.[i]
  printfn "  %d. %s" (i+1) setName
  for j = 0 to testSet.Length - 1 do
    let (d, m, y, expected) = testSet.[j]
    let day = date2Day d m y
    printfn "    test %d - %b" (j+1) (day = expected)
```

Listing 10.5: Output from Listing 10.4.

```
$ fsharpc --nologo date2DayBlackTest.fsx && mono
   date2DayBlackTest.exe
Black-box testing of date2Day.fsx
  1. A complete week
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
    test 5 - true
    test 6 - true
    test 7 - true
  2. Across boundaries
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
  3. Across feburary boundary
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
    test 5 - true
  4. Leap years
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
```

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.** Advice

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.

10.3 Debugging by tracing

Once an error has been found by testing, then the *debugging* phase starts. The cause of a bug can either be that the algorithm chosen is the wrong one for the job, or the implementation of it has an error. In the debugging process, we have to keep an open mind, and not rely on assumptions, since assumptions tend to blind the reader of a text. 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 is 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*, which replaces parts of the code with code that produces safe and · debugging · mockup code relevant results. If the bug is not obvious then more rigorous techniques must be used such as *tracing*. Some development interfaces has built-in tracing system, e.g., `fsharpi` will print inferred types and some binding values. However, often a source of a bug is due to a misunderstanding of the flow of data through a program execution, and we will in the following introduce *hand tracing* a technique to simulate the execution of a program by hand.

· tracing

· hand tracing

Consider the program in Listing 10.6. [5]

Listing 10.6 gcd.fsx:
gcd

```
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
```

The greatest common divisor of 2 integers. which includes a function for calculating the greatest common divisor of 2 integers, and calls this function with the numbers 10 and 15. Hand tracing this program means that we simulate its execution and as part of that keep track of the bindings, assignments and input and output of the program. To do this, we need to consider code snippet's *environment*. E.g., to hand trace the above program, we start by noting the outer environment, called E_0 for short. In line 1, then the `gcd` identifier is bound to a function, hence we write:

· environment

$$
\begin{aligned}
&E_0 :\\
&\quad \text{gcd} \rightarrow \big((a,b), \text{gcd-body}, \varnothing\big)
\end{aligned}
$$

Function bindings like this one is noted as a closure, which is the triplet $\big(\text{arguments}, \text{expression}, \text{environment}\big)$. The closure is everything needed for the expression to be calculated. Here we wrote gcd-body to denote everything after the equal sign in the function binding. Next, F# executes line 9 and 10, and we update our environment to reflect the bindings as,

$$
\begin{aligned}
&E_0 :\\
&\quad \text{gcd} \rightarrow \big((a,b), \text{gcd-body}, \varnothing\big)\\
&\quad a \rightarrow 10\\
&\quad b \rightarrow 15
\end{aligned}
$$

[5] Jon: **This program uses recursion, which has not been introduced yet.**

In line 11 the function is evaluated. This initiates a new environment E_1, and we update our trace as,

$$\begin{aligned}
&E_0: \\
&\quad \text{gcd} \to \big((a,b), \text{gcd-body}, \varnothing\big) \\
&\quad a \to 10 \\
&\quad b \to 15 \\
&\quad \text{line 11: gcd a b} \to ? \\
&E_1: \big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big)
\end{aligned}$$

where the new environment is noted to have gotten its argument names `a` and `b` bound to the values `10` and `15` respectively, and where the return of the function to environment E_0 is yet unknown, so it is noted as a question mark. In line 2 the comparison `a < b` is checked, and since we are in environment E_1 then this is the same as checking `10 < 15`, which is true so the program executes line 3. Hence, we initiate a new environment E_2 and update our trace as,

$$\begin{aligned}
&E_0: \\
&\quad \text{gcd} \to \big((a,b), \text{gcd-body}, \varnothing\big) \\
&\quad a \to 10 \\
&\quad b \to 15 \\
&\quad \text{line 11: gcd a b} \to ? \\
&E_1: \big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 3: gcd b a} \to ? \\
&E_2: \big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big)
\end{aligned}$$

where in the new environment `a` and `b` bound to the values `15` and `10` respectively. In E_2, `10 < 15` is false, so the program evaluates `b > 0`, which is true, hence line 5 is executed. This calls `gcd` once again, but with new arguments, and `a % b` is parenthesized, then it is evaluated before `gcd` is called.

Hence, we update our trace as,

$$\begin{aligned}
&E_0: \\
&\quad \text{gcd} \to \big((a,b), \text{gcd-body}, \varnothing\big) \\
&\quad a \to 10 \\
&\quad b \to 15 \\
&\quad \text{line 11: gcd a b} \to ? \\
&E_1: \big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 3: gcd b a} \to ? \\
&E_2: \big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big) \\
&\quad \text{line 5: a \% b} \to 5 \\
&\quad \text{line 5: gcd b (a \% b)} \to ? \\
&E_3: \big((a \to 10, b \to 5), \text{gcd-body}, \varnothing\big)
\end{aligned}$$

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

E_0 :
  gcd $\rightarrow \big((a, b), \text{gcd-body}, \varnothing\big)$
  $a \rightarrow 10$
  $b \rightarrow 15$
  line 11: gcd a b $\rightarrow$?
E_1 :$\big((a \rightarrow 10, b \rightarrow 15), \text{gcd-body}, \varnothing\big)$
  line 3: gcd b a $\rightarrow$?
E_2 :$\big((a \rightarrow 15, b \rightarrow 10), \text{gcd-body}, \varnothing\big)$
  line 5: a % b $\rightarrow 5$
  line 5: gcd b (a % b) $\rightarrow$?
E_3 :$\big((a \rightarrow 10, b \rightarrow 5), \text{gcd-body}, \varnothing\big)$
  line 5: a % b $\rightarrow 0$
  line 5: gcd b (a % b) $\rightarrow$?
E_4 :$\big((a \rightarrow 5, b \rightarrow 0), \text{gcd-body}, \varnothing\big)$

This time both `a < b` and `b > 0` are false, so we fall through to line 7, and `gcd` from E_4 returns its value of `a`, which is `5`, so we scratch E_4 and change the question mark in E_3 to `5`:

E_0 :
  gcd $\rightarrow \big((a, b), \text{gcd-body}, \varnothing\big)$
  $a \rightarrow 10$
  $b \rightarrow 15$
  line 11: gcd a b $\rightarrow$?
E_1 :$\big((a \rightarrow 10, b \rightarrow 15), \text{gcd-body}, \varnothing\big)$
  line 3: gcd b a $\rightarrow$?
E_2 :$\big((a \rightarrow 15, b \rightarrow 10), \text{gcd-body}, \varnothing\big)$
  line 5: a % b $\rightarrow 5$
  line 5: gcd b (a % b) $\rightarrow$?
E_3 :$\big((a \rightarrow 10, b \rightarrow 5), \text{gcd-body}, \varnothing\big)$
  line 5: a % b $\rightarrow 0$
  line 5: gcd b (a % b) $\rightarrow \cancel{?}\ 5$
$\cancel{E_4}$:$\big((a \rightarrow 5, b \rightarrow 0), \text{gcd-body}, \varnothing\big)$

Now line 5 in E_3 is also a return point of `gcd`, hence we scratch E_3 and change the question mark in

E_2 to 5,

$$
\begin{array}{ll}
E_0: & \\
& \mathrm{gcd} \to \big((a,b), \mathrm{gcd\text{-}body}, \varnothing\big) \\
& a \to 10 \\
& b \to 15 \\
& \text{line 11: gcd a b} \to ? \\
E_1: & \big((a \to 10, b \to 15), \mathrm{gcd\text{-}body}, \varnothing\big) \\
& \text{line 3: gcd b a} \to ? \\
E_2: & \big((a \to 15, b \to 10), \mathrm{gcd\text{-}body}, \varnothing\big) \\
& \text{line 5: a \% b} \to 5 \\
& \text{line 5: gcd b (a \% b)} \to \cancel{?}\, 5 \\
\cancel{E_3}: & \big((a \to 10, b \to 5), \mathrm{gcd\text{-}body}, \varnothing\big) \\
& \text{line 5: a \% b} \to 0 \\
& \text{line 5: gcd b (a \% b)} \to \cancel{?}\, 5 \\
\cancel{E_4}: & \big((a \to 5, b \to 0), \mathrm{gcd\text{-}body}, \varnothing\big)
\end{array}
$$

and likewise, for E_2 and E_1:

$$
\begin{array}{ll}
E_0: & \\
& \mathrm{gcd} \to \big((a,b), \mathrm{gcd\text{-}body}, \varnothing\big) \\
& a \to 10 \\
& b \to 15 \\
& \text{line 11: gcd a b} \to \cancel{?}\, 5 \\
\cancel{E_1}: & \big((a \to 10, b \to 15), \mathrm{gcd\text{-}body}, \varnothing\big) \\
& \text{line 3: gcd b a} \to \cancel{?}\, 5 \\
\cancel{E_2}: & \big((a \to 15, b \to 10), \mathrm{gcd\text{-}body}, \varnothing\big) \\
& \text{line 5: a \% b} \to 5 \\
& \text{line 5: gcd b (a \% b)} \to \cancel{?}\, 5 \\
\cancel{E_3}: & \big((a \to 10, b \to 5), \mathrm{gcd\text{-}body}, \varnothing\big) \\
& \text{line 5: a \% b} \to 0 \\
& \text{line 5: gcd b (a \% b)} \to \cancel{?}\, 5 \\
\cancel{E_4}: & \big((a \to 5, b \to 0), \mathrm{gcd\text{-}body}, \varnothing\big)
\end{array}
$$

Now we are able to continue the program in environment E_0 with the `printfn` statement, and we write:

$$
\begin{aligned}
&E_0:\\
&\quad \text{gcd} \to \big((a,b), \text{gcd-body}, \varnothing\big)\\
&\quad a \to 10\\
&\quad b \to 15\\
&\quad \text{line 11: gcd a b} \to \cancel{?}\,5\\
&\quad \text{line 11: stdout} \to \text{"gcd 10 15} = \text{5"}\\
&\cancel{E_1}:\big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big)\\
&\quad \text{line 3: gcd b a} \to \cancel{?}\,5\\
&\cancel{E_2}:\big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big)\\
&\quad \text{line 5: a \% b} \to 5\\
&\quad \text{line 5: gcd b (a \% b)} \to \cancel{?}\,5\\
&\cancel{E_3}:\big((a \to 10, b \to 5), \text{gcd-body}, \varnothing\big)\\
&\quad \text{line 5: a \% b} \to 0\\
&\quad \text{line 5: gcd b (a \% b)} \to \cancel{?}\,5\\
&\cancel{E_4}:\big((a \to 5, b \to 0), \text{gcd-body}, \varnothing\big)
\end{aligned}
$$

which completes the hand tracing of `gcd.fsx`.

F# uses lexical scope, which implies that besides function arguments, we also at times need to consider the environment at place of writing. Consider the program in Listing 10.7.

**Listing 10.7 lexicalScopeTracing.fsx:
lexicalScopeTracing**

```
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)
```

```
$ fsharpc --nologo lexicalScopeTracing.fsx
$ mono lexicalScopeTracing.exe
6.0
```

Example of lexical scope and closure environment. To hand trace this, we start by creating the outer environment, define the closure for `testScope`, and reach line 6,

$$
\begin{aligned}
&E_0:\\
&\quad \text{testScope} \to \big(x, \text{testScope-body}, \varnothing\big)\\
&\quad \text{line 6: testScope 2.0} \to ?
\end{aligned}
$$

We create new environment for `testScope` and note the bindings,

$$
\begin{array}{ll}
E_0: & \\
 & \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big) \\
 & \text{line 6: testScope 2.0} \rightarrow ? \\
E_1: & \big(x \rightarrow 2.0, \text{testScope-body}, \varnothing\big) \\
 & a \rightarrow 3.0 \\
 & \text{f} \rightarrow \big(z, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0)\big) \\
 & a \rightarrow 4.0
\end{array}
$$

Since we are working with lexical scope, then `a` is noted twice, and its interpretation is by lexical order. Hence, the environment for the closure of `f` is everything above in E_1, so we add $a \rightarrow 3.0$ and $x \rightarrow 2.0$. In line 5 `f` is called, so we create an environment based on its closure,

$$
\begin{array}{ll}
E_0: & \\
 & \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big) \\
 & \text{line 6: testScope 2.0} \rightarrow ? \\
E_1: & \big(x \rightarrow 2.0, \text{testScope-body}, \varnothing\big) \\
 & a \rightarrow 3.0 \\
 & \text{f} \rightarrow \big(z, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0)\big) \\
 & a \rightarrow 4.0 \\
 & \text{line 5: f x} \rightarrow ? \\
E_2: & \big(z \rightarrow 2.0, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0\big)
\end{array}
$$

The expression in the environment E_2 evaluates to `6.0`, and unravelling the scopes we get,

$$
\begin{array}{ll}
E_0: & \\
 & \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big) \\
 & \text{line 6: testScope 2.0} \rightarrow \cancel{?}\, 6.0 \\
 & \text{line 6: stdout} \rightarrow \texttt{"6.0"} \\
\cancel{E_1}: & \big(x \rightarrow 2.0, \text{testScope-body}, \varnothing\big) \\
 & a \rightarrow 3.0 \\
 & \text{f} \rightarrow \big(z, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0)\big) \\
 & a \rightarrow 4.0 \\
 & \text{line 5: f x} \rightarrow \cancel{?}\, 6.0 \\
\cancel{E_2}: & \big(z \rightarrow 2.0, \text{a * z}, (a \rightarrow 3.0, x \rightarrow 2.0\big)
\end{array}
$$

For mutable bindings, i.e., variables, the scope is dynamic. For this we need the concept of storage. Consider the program in Listing 10.8.

Listing 10.8 dynamicScopeTracing.fsx: dynamicScopeTracing

```
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)
```

```
$ fsharpc --nologo dynamicScopeTracing.fsx
$ mono dynamicScopeTracing.exe
8.0
```

Example of lexical scope and closure environment. We add a storage area to our hand tracing, e.g., line 6,

$$
\begin{aligned}
&\text{Store}:\\
&\quad E_0:\\
&\qquad \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big)\\
&\qquad \text{line 6: testScope 2.0} \rightarrow ?
\end{aligned}
$$

So when we generate environment E_1, the mutable binding is to a storage location,

$$
\begin{aligned}
&\text{Store}:\\
&\qquad \alpha_1 \rightarrow 3.0\\
&\quad E_0:\\
&\qquad \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big)\\
&\qquad \text{line 6: testScope 2.0} \rightarrow ?\\
&\quad E_1: \big(x \rightarrow 2.0, \text{testScope-body}, \varnothing\big)\\
&\qquad a \rightarrow \alpha_1
\end{aligned}
$$

which is assigned the value `3.0` at the definition of `a`. Now the definition of `f` is uses the storage location

$$
\begin{aligned}
&\text{Store}:\\
&\qquad \alpha_1 \rightarrow 3.0\\
&\quad E_0:\\
&\qquad \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big)\\
&\qquad \text{line 6: testScope 2.0} \rightarrow ?\\
&\quad E_1: \big(x \rightarrow 2.0, \text{testScope-body}, \varnothing\big)\\
&\qquad a \rightarrow \alpha_1\\
&\qquad \text{f} \rightarrow \big(z, \text{a * z}, (a \rightarrow \alpha_1, x \rightarrow 2.0)\big)
\end{aligned}
$$

and in line 4 it is the value in the storage, which is updated,

$$
\begin{aligned}
&\text{Store :}\\
&\qquad \alpha_1 \to \cancel{3.0}\, 4.0\\
&\quad E_0 :\\
&\qquad \text{testScope} \to \big(x, \text{testScope-body}, \varnothing\big)\\
&\qquad \text{line 6: testScope } 2.0 \to ?\\
&\quad E_1 : \big(x \to 2.0, \text{testScope-body}, \varnothing\big)\\
&\qquad a \to \alpha_1\\
&\qquad \text{f} \to \big(z, \text{a * z}, (a \to \alpha_1, x \to 2.0)\big)
\end{aligned}
$$

Hence,

$$
\begin{aligned}
&\text{Store :}\\
&\qquad \alpha_1 \to \cancel{3.0}\, 4.0\\
&\quad E_0 :\\
&\qquad \text{testScope} \to \big(x, \text{testScope-body}, \varnothing\big)\\
&\qquad \text{line 6: testScope } 2.0 \to ?\\
&\quad E_1 : \big(x \to 2.0, \text{testScope-body}, \varnothing\big)\\
&\qquad a \to \alpha_1\\
&\qquad \text{f} \to \big(z, \text{a * z}, (a \to \alpha_1, x \to 2.0)\big)\\
&\qquad \text{line 5: f x} \to ?\\
&\quad E_2 : \big(z \to 2.0, \text{a * z}, (a \to \alpha_1, x \to 2.0\big)
\end{aligned}
$$

and the return value from `f` evaluated in environment E_2 now reads the value `4.0` for `a` and returns `8.0`. Hence,

$$
\begin{aligned}
&\text{Store :}\\
&\qquad \alpha_1 \to \cancel{3.0}\, 4.0\\
&\quad E_0 :\\
&\qquad \text{testScope} \to \big(x, \text{testScope-body}, \varnothing\big)\\
&\qquad \text{line 6: testScope } 2.0 \to \cancel{?}\, 8.0\\
&\qquad \text{line 6: stdout} \to \texttt{"8.0"}\\
&\quad \cancel{E_1} : \big(x \to 2.0, \text{testScope-body}, \varnothing\big)\\
&\qquad a \to \alpha_1\\
&\qquad \text{f} \to \big(z, \text{a * z}, (a \to \alpha_1, x \to 2.0)\big)\\
&\qquad \text{line 5: f x} \to \cancel{?}\, 8.0\\
&\quad \cancel{E_2} : \big(z \to 2.0, \text{a * z}, (a \to \alpha_1, x \to 2.0\big)
\end{aligned}
$$

As can be seen by the above examples, hand tracing can be used to in detail study the flow of data through a program. It may seem tedious in the beginning, but the care illustrated above is useful at start to ensure rigor in the analysis. Most will find that once accustomed to the method, the analysis can be performed rigorously but with less paperwork, and in conjunction with strategically placed debugging `printfn` statements, it is a very valuable tool for debugging.

11 | 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, arrays, and sequences. Strings were discussed in Chapter 5 and will be revisited here in more details. Sequences will not be discussed,[1] and we will concentrate on lists and one- and two-dimensional arrays.

11.1 Strings

Strings have been discussed in Chapter 5, 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 Appendix C for further details on the unicode standard. The type `string` is an alias for *`System.string`*. String literals are delimited by quotation marks "`"`" and inside the delimiters, character escape sequences are allowed (see Table 5.2), 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 a double quotation marks is an escape sequence which is replaced by a single, e.g., `@"This is a string"`, `@"\tNon-tabulated string"`, `@"A ""quoted"" string"`, and `@""`. Alternatively, a verbatim string may be delimited by tripple quotes, e.g., `"""This is a string"""`, `"""\tNon-tabulated string"""`, `"""A "quoted" string"""`, and `""""""`. Strings may be indexed using the `.[]` notation, as demonstrated in Listing 5.27.

· string
· `System.string`
· verbatim string

11.1.1 String properties

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

`Length:` Returns the length of the string. Compare with `String.length` method.

Listing 11.1: `Length`

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

[1] Jon: **Should we discuss sequences?**

11.1.2 String module

In the `String` module the following functions are available.

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

Listing 11.2: `String.collect`

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

`String.concat: string -> seq<string> -> string`. Returns a new string made by concatenating the given strings with a separator. Here `seq<string>` is a sequence but can also be a list or an array.

Listing 11.3: `String.concat`

```
> String.concat ", " ["abc"; "def"; "ghi"];;
val it : string = "abc, def, ghi"
```

`String.exists: (char -> bool) -> string -> bool`. Tests if any character of the string satisfies the given predicate.

Listing 11.4: `String.exists`

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

`String.forall: (char -> bool) -> string -> bool`. Tests if all characters in the string satisfy the given predicate.

Listing 11.5: `String.forall`

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

`String.init: int -> (int -> string) -> string`. Creates a new string whose characters are the results of applying a specified function to each index and concatenating the resulting strings.

Listing 11.6: `String.init`

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

`String.iter: (char -> unit) -> string -> unit`. Applies a specified function to each character in the string.

Listing 11.7: `String.iter`

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

`String.iteri: (int -> char -> unit) -> string -> unit`. Applies a specified function to the index of each character in the string and the character itself.

Listing 11.8: `String.iteri`

```
> String.iteri (fun i c -> printfn "%d: %c" i c) "abc";;
0: a
1: b
2: c
val it : unit = ()
```

`String.length: string -> int`. Returns the length of the string.

Listing 11.9: `String.length`

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

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

Listing 11.10: `String.map`

```
> let dc = int 'A' - int 'a'
- String.map (fun c -> c + char dc) "abcd";;
val dc : int = -32
val it : string = "ABCD"
```

`String.mapi: (int -> char -> char) -> string -> string`. Creates a new string whose characters are the results of applying a specified function to each character and index of the input string.

Listing 11.11: `String.mapi`

```
> String.mapi (fun i c -> char (int c + i)) "aaaa";;
val it : string = "abcd"
```

`String.replicate: int -> string -> string`. Returns a string by concatenating a specified number of instances of a string.

Listing 11.12: `String.replicate`

```
> String.replicate 4 "abc, ";;
val it : string = "abc, abc, abc, abc, "
```

11.2 Lists

Lists are unions of immutable values of the same type and have a more flexible structure than tuples. Lists can be expressed as a *sequence expression*,

· list
· sequence expression

Listing 11.13 Lists with a sequence expression.

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

Examples are `[1; 2; 3; 4; 5]`, which represents a list of integers, `["This"; "is"; "a"; "list"]`, which represents a list of strings, `[(fun x -> x); (fun x -> x*x)]`, which represents a list of anonymous functions, and `[]`, which is an empty list. Lists may also be given as ranges,

Listing 11.14 Lists with a 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]`.

· range expressions

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 using the *`isEmpty`* property. These features are demonstrated in Listing 11.15.

· `list`
· `.[]`
· `Length`
· `isEmpty`

Listing 11.15 listIndexing.fsx:
Lists are indexed as strings and has a `Length` property.

```
let printList (lst : int list) : unit =
  for i = 0 to lst.Length - 1 do
    printf "%A " lst.[i]
  printfn ""

let lst = [3; 4; 5]
printfn "lst = %A, lst.[1] = %A" lst lst.[1]
printfn "lst.Length = %A, lst.isEmpty = %A" lst.Length lst.IsEmpty
printList lst
```

```
$ fsharpc --nologo listIndexing.fsx && mono listIndexing.exe
lst = [3; 4; 5], lst.[1] = 4
lst.Length = 3, lst.isEmpty = false
3 4 5
```

F# implements lists as linked lists, see Figure 11.1, which is why indexing element i has computational complexity $\mathcal{O}(i)$, since the list has to be traversed from the beginning until element i is located. Thus, **indexing lists is slow and should be avoided.**

Advice

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,

· for
· `in`

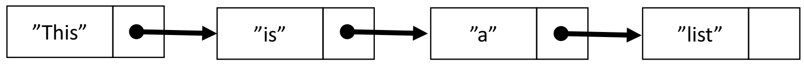


Figure 11.1: A list is a linked list: Here is illustrated the linked list of `["This"; "is"; "a"; "list"]`.

Listing 11.16 For-in loop with in expression.

```
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 Listing 11.17.

**Listing 11.17 listFor.fsx:
The `for-in` loops are preferred over `for-to`.**

```
let printList (lst : int list) : unit =
  for elm in lst do
    printf "%A " elm
  printfn ""

printList [3; 4; 5]
```

```
$ fsharpc --nologo listFor.fsx && mono listFor.exe
3 4 5
```

Using `for-in`-expressions remove the risk of off-by-one indexing errors, and thus, **`for-in` is to be preferred over `for-to`.** Advice

Lists support slicing identically to strings as demonstrated in Listing 11.18.

**Listing 11.18 listSlicing.fsx:
Examples of list slicing. Compare with Listing 5.27.**

```
let lst = ['a' .. 'g']
printfn "%A" lst.[0]
printfn "%A" lst.[3]
printfn "%A" lst.[3..]
printfn "%A" lst.[..3]
printfn "%A" lst.[1..3]
printfn "%A" lst.[*]
```

```
$ fsharpc --nologo listSlicing.fsx && mono listSlicing.exe
'a'
'd'
['d'; 'e'; 'f'; 'g']
['a'; 'b'; 'c'; 'd']
['b'; 'c'; 'd']
['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g']
```

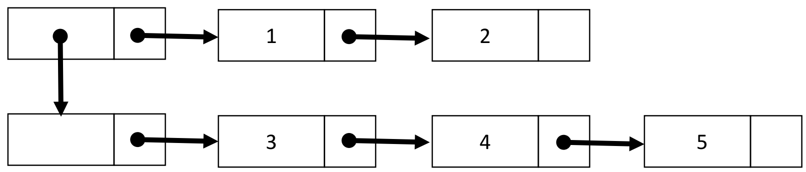


Figure 11.2: A list is a ragged linked list: Here is illustrated the linked list of `[[1;2];[3;4;5]]`.

Lists may be concatenated using either the *"`@`"*[2] *concatenation* operator or the *"`::`" cons* operators. The differences is that "`@`" concatenates two lists of identical types, while "`::`" concatenates an element and a list of identical types. This is demonstrated in Listing 11.19.

· `@`
· list concatenation
· `::`
· list cons

Listing 11.19 listCon.fsx:
Examples of list concatenation.

```
printfn "[1] @ [2; 3] = %A" ([1] @ [2; 3])
printfn "[1; 2] @ [3; 4] = %A" ([1; 2] @ [3; 4])
printfn "1 :: [2; 3] = %A" (1 :: [2; 3])
```

```
$ fsharpc --nologo listCon.fsx && mono listCon.exe
[1] @ [2; 3] = [1; 2; 3]
[1; 2] @ [3; 4] = [1; 2; 3; 4]
1 :: [2; 3] = [1; 2; 3]
```

Since lists are represented as linked lists, then 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 Listing 11.20.

Listing 11.20 listMultidimensional.fsx:
A ragged multidimensional list, built as lists of lists, and its indexing.

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

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

The example shows a *ragged multidimensional list*, since each row has a different number of elements. This is also illustrated in Figure 11.2.

· ragged multidimensional list

The indexing of a particular element is slow due to the linked list implementation of lists, which is

[2]Jon: **why does the at-symbol not appear in the index?**

why arrays are often preferred for two- and higher-dimensional data structures, see Section 11.3.

11.2.1 List properties

Lists supports a number of properties, i.e., values that are attached to each list and access using the "." notation, some of which are:

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

Listing 11.21: `Head`

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

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

Listing 11.22: `Head`

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

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

Listing 11.23: `Length`

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

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

Listing 11.24: `Tail`

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

11.2.2 List module

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

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

Listing 11.25: `List.collect`

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

`List.contains: 'T -> 'T list -> bool`. Returns true or false depending on whether an element is contained in the list.

Listing 11.26: `List.contains`

```
> List.contains 3 [1; 2; 3];;
val it : bool = true
```

`List.empty: 'T list`. An empty list of inferred type.

Listing 11.27: `List.empty`

```
> let a : int list = List.empty;;
val a : int list = []
```

`List.exists: ('T -> bool) -> 'T list -> bool`. Returns true or false depending on whether any element is true for a given function.

Listing 11.28: `List.exists`

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

`List.filter: ('T -> bool) -> 'T list -> 'T list`. Returns a new list, of all the elements of the original list for which the supplied function evaluates to true.

Listing 11.29: `List.filter`

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

`List.find: ('T -> bool) -> 'T list -> 'T`. Return the first element for which the given function is true.

Listing 11.30: `List.find`

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

`List.findIndex: ('T -> bool) -> 'T list -> int`. Return the index of the first element for which the given function is true.

Listing 11.31: `List.findIndex`

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

`List.fold: ('State -> 'T -> 'State) -> 'State -> 'T list -> 'State`. Update an accumulator iteratively by applying the supplied function to each element in a list, e.g. for a list consisting of $x_0, x_1, x_2, \ldots, x_n$, a supplied function f, and an initial value for the accumulator s, calculate $f(\ldots f(f(f(s, x_0), x_1), x_2), \ldots, x_n)$.

Listing 11.32: `List.fold`

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

`List.foldBack: ('T -> 'State -> 'State) -> 'T list -> 'State -> 'State`. Update an accumulator iteratively by applying function to each element in a list, e.g. for a list consisting of $x_0, x_1, x_2, \ldots, x_n$, a supplied function f, and an initial value for the accumulator s, calculate $f(x_0, f(x_1, f(x_2, \ldots, f(x_n, s))))$.

Listing 11.33: `List.foldBack`

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

`List.forall: ('T -> bool) -> 'T list -> bool`. Apply a function to all element and logically and the result.

Listing 11.34: `List.forall`

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

`List.head: 'T list -> int`. The first element in the list. Exception if empty.

Listing 11.35: `List.head`

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

`List.isEmpty: 'T list -> int`. Compare with the empty list

Listing 11.36: `List.isEmpty`

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

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

`List.item: 'T list -> int`. Retrieve an element of a list by its index.

Listing 11.37: `List.item`

```
> let a = [1; -3; 0] in List.item 1 a;;
val it : int = -3
```

`List.iter: ('T -> unit) -> 'T list -> unit`. Apply a procedure to every element in the list.

Listing 11.38: `List.iter`

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

`List.Length: 'T list -> int`. The number of elements in a list

Listing 11.39: `List.Length`

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

`List.map: ('T -> 'U) -> 'T list -> 'U list`. Return a list, where the supplied function has been applied to every element.

Listing 11.40: `List.map`

```
> let square x = x*x in List.map square [0 .. 9];;
val it : int list = [0; 1; 4; 9; 16; 25; 36; 49; 64; 81]
```

`List.ofArray: 'T list -> int`. Return a list whose elements are the same as the supplied array.

Listing 11.41: `List.ofArray`

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

`List.rev: 'T list -> 'T list`. Return a list whose elements have been reversed.

Listing 11.42: `List.rev`

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

`List.sort: 'T list -> 'T list`. Return a list whos elements have been sorted.

Listing 11.43: `List.sort`

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

`List.tail: 'T list -> 'T list`. The list except its first element. Exception if empty.

Listing 11.44: `List.tail`

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

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

`List.toArray: 'T list -> 'T []`. Return an array whos elements are the same as the supplied list.

Listing 11.45: `List.toArray`

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

`List.unzip: ('T1 * 'T2) list -> 'T1 list * 'T2 list`. Return a pair of lists, whos elements are take from pairs of a list.

Listing 11.46: `List.unzip`

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

>
```

`List.zip: 'T1 list -> 'T2 list -> ('T1 * 'T2) list`. Return a list of pairs, whos elements are take iteratively from two lists.

Listing 11.47: `List.zip`

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

11.3 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 *sequence expressions*,

· arrays
· sequence expressions

Listing 11.48 Arrays with a sequence expression.

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

and examples are `[|1; 2; 3; 4; 5|]`, which is an array of integers, `[|"This"; "is"; "an"; "array"|]`, which is an array of strings, `[|(fun x -> x); (fun x -> x*x)|]`, which is an array of anonymous functions, and `[||]`, which is an empty array. Arrays may also be given as ranges,

Listing 11.49 Arrays with a range expressions.

```
[|<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|]`. · range expressions

The array type is defined using the **array** keyword or alternatively the "`[]`" lexeme. Like strings and lists, arrays may be indexed using the "*.[]*" notation. Arrays cannot be resized, but are mutable as shown in Listing 11.50. · .[]

Listing 11.50 arrayReassign.fsx:
Arrays are mutable in spite 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|]
```

Notice that in spite the missing `mutable` keyword, the function **square** still had the *side-effect* of squaring alle entries in `A`. F# implements arrays as chunks of memory and indexes arrays via adresse arithmetic. I.e., element i in an array, whose first element is in memory adresse α and whose elements fill β adresses each is found at adresse $\alpha + i\beta$.[3] 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.** · side-effect Advice

Arrays support *slicing*, that is, indexing an array with a range results in a copy of array with values corresponding to the range. This is demonstrated in Listing 11.51. · slicing

[3]Jon: **Add a figure illustrating adress indexing.**

Listing 11.51 arraySlicing.fsx:
Examples of array slicing. Compare with Listing 11.18 and Listing 5.27.

```
let arr = [|'a' .. 'g'|]
printfn "%A" arr.[0]
printfn "%A" arr.[3]
printfn "%A" arr.[3..]
printfn "%A" arr.[..3]
printfn "%A" arr.[1..3]
printfn "%A" arr.[*]
```

```
$ fsharpc --nologo arraySlicing.fsx && mono arraySlicing.exe
'a'
'd'
[|'d'; 'e'; 'f'; 'g'|]
[|'a'; 'b'; 'c'; 'd'|]
[|'b'; 'c'; 'd'|]
[|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'|]
```

As illustrated, the missing start or end index implies from the first or to the last element.

Arrays have explicit operator support for appending and concatenation, instead the `Array` namespace includes an `Array.append` function, as shown in Listing 11.52.

Listing 11.52 arrayAppend.fsx:
Two arrays are appended with `Array.append`.

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

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

Arrays are *reference types*, meaning that identifiers are references and thus suffers from aliasing, as illustrated in Listing 11.53.

· reference types

Listing 11.53 arrayAliasing.fsx:
Arrays are reference types and suffers from aliasing.

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

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

11.3.1 Array properties and methods

Arrays supports a number of properties and methods, i.e., values and functions that are attached to each array and access using the "`.`" notation, some of which are:

`Clone()`: Returns a copy of the array.

Listing 11.54: `Clone`

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

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

Listing 11.55: `Length`

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

11.3.2 Array module

There are quite a number of built-in procedures for arrays in the `Array` module, some of which are summarized below.[4]

`Array.append`: `'T [] -> 'T [] -> 'T []`. Creates an array that contains the elements of one array followed by the elements of another array.

Listing 11.56: `Array.append`

```
> Array.append [|1; 2;|] [|3; 4; 5|];;
val it : int [] = [|1; 2; 3; 4; 5|]
```

`Array.compareWith`: `('T -> 'T -> int) -> 'T [] -> 'T [] -> int`. Compares two arrays using the given comparison function, element by element.

Listing 11.57: `Array.compareWith`

```
> let compArr elm1 elm2 =
-     if elm1 > elm2 then 1
-     elif elm1 < elm2 then -1
-     else 0
- Array.compareWith compArr [|1; 2; 4|] [|1; 2; 3|];;
val compArr : elm1:'a -> elm2:'a -> int when 'a : comparison
val it : int = 1
```

[4] Jon: **rewrite description**

`Array.concat: seq<'T []> -> 'T []`. Creates an array that contains the elements of each of the supplied sequence of arrays.

Listing 11.58: `Array.concat`

```
> Array.concat [[|1; 2; 3|]; [|4; 5|]; [|6; 7; 8|]];;
val it : int [] = [|1; 2; 3; 4; 5; 6; 7; 8|]
```

`Array.contains:` . Evaluates to true if the given element is in the input array.

Listing 11.59: `Array.contains`

```
> Array.contains 3 [|1; 2; 3|];;
val it : bool = true
```

`Array.copy: 'T [] -> 'T []`. Creates an array that contains the elements of the supplied array.

Listing 11.60: `Array.copy`

```
> let a = [|1; 2; 3|]
- let b = Array.copy a;;
val a : int [] = [|1; 2; 3|]
val b : int [] = [|1; 2; 3|]
```

`Array.create: int -> 'T -> 'T []`. Creates an array whose elements are initiallized the supplied value.

Listing 11.61: `Array.create`

```
> Array.create 4 3.14;;
val it : float [] = [|3.14; 3.14; 3.14; 3.14|]
```

`Array.empty: 'T []`. Returns an empty array of the given type.

Listing 11.62: `Array.empty`

```
> let a : int [] = Array.empty;;
val a : int [] = [||]
```

`Array.exists: ('T -> bool) -> 'T [] -> bool`. Tests whether any element of an array satisfies the supplied predicate.

Listing 11.63: `Array.exists`

```
> let odd x = (x % 2 = 1) in Array.exists odd [|0 .. 2 .. 4|];;
val it : bool = false
```

`Array.fill: 'T [] -> int -> int -> 'T -> unit`. Fills a range of elements of an array with the supplied value.

Listing 11.64: `Array.fill`

```
> let arr = Array.zeroCreate 10;
- Array.fill arr 2 5 2;;
val arr : int [] = [|0; 0; 2; 2; 2; 2; 2; 0; 0; 0|]
val it : unit = ()
```

`Array.filter: ('T -> bool) -> 'T [] -> 'T []`. Returns a collection that contains only the elements of the supplied array for which the supplied condition returns true.

Listing 11.65: `Array.filter`

```
> let odd x = (x % 2 = 1) in Array.filter odd [|0 .. 9|];;
val it : int [] = [|1; 3; 5; 7; 9|]
```

`Array.find: ('T -> bool) -> 'T [] -> 'T`. Returns the first element for which the supplied function returns true. Raises `System.Collections.Generic.KeyNotFoundException`

Listing 11.66: `Array.find`

```
> let odd x = (x % 2 = 1) in Array.find odd [|0 .. 9|];;
val it : int = 1
```

`Array.findIndex: ('T -> bool) -> 'T [] -> int`. Returns the index of the first element in an array that satisfies the supplied condition. Raises `System.Collections.Generic.KeyNotFoundException` if none of the elements satisfy the condition.

Listing 11.67: `Array.findIndex`

```
> let isK x = (x = 'k') in Array.findIndex isK [|'a' .. 'z'|];;
val it : int = 10
```

`Array.fold: ('State -> 'T -> 'State) -> 'State -> 'T [] -> 'State`. Applies a function to each element of an array, threading an accumulator argument through the computation. If the input function is f and the array elements are i0...iN, this function computes f (...(f s i0)...) iN.

Listing 11.68: `Array.fold`

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

`Array.foldBack: ('T -> 'State -> 'State) -> 'T [] -> 'State -> 'State`. Applies a function to each element of an array, threading an accumulator argument through the computation. If the input function is f and the array elements are i0...iN, this function computes f i0 (...(f iN s)).

Listing 11.69: `Array.foldBack`

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

`Array.forall: ('T -> bool) -> 'T [] -> bool`. Tests whether all elements of an array satisfy the supplied condition.

Listing 11.70: `Array.forall`

```
> let odd x = (x % 2 = 1) in Array.forall odd [|0 .. 9|];;
val it : bool = false
```

`Array.get: 'T [] -> int -> 'T`. Gets an element from an array.

Listing 11.71: `Array.get`

```
> Array.get [|1; 2; 3|] 2;;
val it : int = 3
```

`Array.init: int -> (int -> 'T) -> 'T []`. Uses a supplied function to create an array of the supplied dimension.

Listing 11.72: `Array.init`

```
> let squareIndex ind = ind*ind
- Array.init 10 squareIndex;;
val squareIndex : ind:int -> int
val it : int [] = [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

`Array.isEmpty: 'T [] -> bool`. Tests whether an array has any elements.

Listing 11.73: `Array.isEmpty`

```
> Array.isEmpty [||];;
val it : bool = true
```

`Array.iter: ('T -> unit) -> 'T [] -> unit`. Applies the supplied function to each element of an array.

Listing 11.74: `Array.iter`

```
> let prt x = printfn "%A " x in Array.iter prt [|0; 1; 2|];;
0
1
2
val it : unit = ()
```

`Array.length: 'T [] -> int`. Returns the length of an array. The `System.Array.Length` property does the same thing.

Listing 11.75: `Array.length`

```
> let a = [|1; 2; 3|] in a.Length;;
val it : int = 3
```

`Array.map: ('T -> 'U) -> 'T [] -> 'U []`. Creates an array whose elements are the results of applying the supplied function to each of the elements of a supplied array.

Listing 11.76: `Array.map`

```
> let square x = x*x in Array.map square [|0 .. 9|];;
val it : int [] = [|0; 1; 4; 9; 16; 25; 36; 49; 64; 81|]
```

`Array.ofList: 'T list -> 'T []`. Creates an array from the supplied list.

Listing 11.77: `Array.ofList`

```
> Array.ofList [1; 2; 3];;
val it : int [] = [|1; 2; 3|]
```

`Array.rev: 'T [] -> 'T []`. Reverses the order of the elements in a supplied array.

Listing 11.78: `Array.rev`

```
> Array.rev [|1; 2; 3|];;
val it : int [] = [|3; 2; 1|]
```

`Array.set: 'T [] -> int -> 'T -> unit`. Sets an element of an array.

Listing 11.79: `Array.set`

```
> let arr = [|1; 2; 3|]
- Array.set arr 2 10
- printfn "%A" arr;;
[|1; 2; 10|]
val arr : int [] = [|1; 2; 10|]
val it : unit = ()
```

`Array.sort: 'T[] -> 'T []`. Sorts the elements of an array and returns a new array. `Operators.compare` is used to compare the elements.

Listing 11.80: `Array.sort`

```
> Array.sort [|3; 1; 2|];;
val it : int [] = [|1; 2; 3|]
```

`Array.sub: 'T [] -> int -> int -> 'T []`. Creates an array that contains the supplied subrange, which is specified by starting index and length.

Listing 11.81: `Array.sub`

```
> Array.sub [|0..9|] 2 5;;
val it : int [] = [|2; 3; 4; 5; 6|]
```

`Array.toList: 'T [] -> 'T list`. Converts the supplied array to a list.

Listing 11.82: `Array.toList`

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

`Array.unzip: ('T1 * 'T2) [] -> 'T1 [] * 'T2 []`. Splits an array of tuple pairs into a tuple of two arrays.

Listing 11.83: `Array.unzip`

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

`Array.zip: 'T1 [] -> 'T2 [] -> ('T1 * 'T2) []`. Combines three arrays into an array of tuples that have three elements. The three arrays must have equal lengths; otherwise, `System.ArgumentException` is raised.

Listing 11.84: `Array.zip`

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

11.4 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. E.g., the example in Listing 11.85 is a jagged array of increasing width.

· multidimensional arrays
· jagged arrays

Listing 11.85 arrayJagged.fsx:
An array of arrays. When row lengths are of non-equal elements, then it is a Jagged array.

```
let arr = [|[|1|]; [|1; 2|]; [|1; 2; 3|]|]

for row in arr do
  for elm in row do
    printf "%A " elm
  printf "\n"
```

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

Indexing 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 Listing 11.86.

Listing 11.86 arrayJaggedSquare.fsx:
A rectangular array.

```
let pownArray  (arr : int array array) p =
  for i = 1 to arr.Length - 1 do
    for j = 1 to arr.[i].Length - 1 do
      arr.[i].[j] <- pown arr.[i].[j] p

let printArrayOfArrays (arr : int array array) =
  for row in arr do
    for elm in row do
      printf "%3d " elm
    printf "\n"

let A = [|[|1 .. 4|]; [|1 .. 2 .. 7|]; [|1 .. 3 .. 10|]|]
pownArray A 2
printArrayOfArrays A
```

```
$ fsharpc --nologo arrayJaggedSquare.fsx && mono arrayJaggedSquare.exe
  1   2   3   4
  1   9  25  49
  1  16  49 100
```

Notice, 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. In fact, square arrays of dimensions 2 to 4 are so common that F# has built-in modules for their support. In the following, we describe *`Array2D`*. The workings of *`Array3D`* and *`Array4D`* are very similar. An example of creating the same 2-dimensional array as above but as an `Array2D` is shown in Listing 11.87.

· `Array2D`
· `Array3D`
· `Array4D`

Listing 11.87 array2D.fsx:
Creating a 3 by 4 rectangular arrays of intigers.

```
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]]
```

Notice that the indexing uses a slightly different notation `[,]` and the length functions are also slightly different. The statement `A.Length` would return the total number of elements in the array, in this case 12. As can be seen, the `printf` supports direct printing of the 2 dimensional array. Higher dimensional arrays support slicing as shown in Listing 11.88.

Listing 11.88 array2DSlicing.fsx:
Examples of Array2D slicing. Compare with Listing 11.87.

```
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.[2,3]
printfn "%A" arr.[1..,3..]
printfn "%A" arr.[..1,*]
printfn "%A" arr.[1,*]
printfn "%A" arr.[1..1,*]
```

```
$ fsharpc --nologo array2DSlicing.fsx && mono array2DSlicing.exe
11
[[10]
 [11]]
[[0; 3; 6; 9]
 [1; 4; 7; 10]]
[|1; 4; 7; 10|]
[[1; 4; 7; 10]]
```

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. [5]

Multidimensional arrays have the same properties and methods as arrays, see Section 11.3.1.

[5]Jon: **Array2D.ToString produces `[[ ... ]]` and not `[|[| ... |]|]`, which can cause confusion.**

11.4.1 Array2D module

There are quite a number of built-in procedures for arrays in the `Array2D` namespace, some of which are summarized below.[6]

`copy: 'T [,] -> 'T [,]`. Creates a new array whose elements are the same as the input array.

Listing 11.89: `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: int -> int -> 'T -> 'T [,]`. Creates an array whose elements are all initially the given value.

Listing 11.90: `Array2D.create`

```
> Array2D.create 2 3 3.14;;
val it : float [,] = [[3.14; 3.14; 3.14]
                      [3.14; 3.14; 3.14]]
```

`get: 'T [,] -> int -> int -> 'T`. Fetches an element from a 2D array. You can also use the syntax `array.[index1,index2]`.

Listing 11.91: `Array2D.get`

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j)
- Array2D.get arr 1 2;;
val arr : int [,] = [[0; 10; 20; 30]
                     [1; 11; 21; 31]
                     [2; 12; 22; 32]]
val it : int = 21
```

`init: int -> int -> (int -> int -> 'T) -> 'T [,]`. Creates an array given the dimensions and a generator function to compute the elements.

Listing 11.92: `Array2D.init`

```
> let idxFct i j = i + 10 * j
- Array2D.init 3 4 idxFct;;
val idxFct : i:int -> j:int -> int
val it : int [,] = [[0; 10; 20; 30]
                    [1; 11; 21; 31]
                    [2; 12; 22; 32]]
```

`iter: ('T -> unit) -> 'T [,] -> unit`. Applies the given function to each element of the array.

[6] Jon: **rewrite description**

Listing 11.93: `Array2D.iter`

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j)
- Array2D.iter (fun elm -> printf "%A " elm) arr
- printfn "";;
0 10 20 30 1 11 21 31 2 12 22 32
val arr : int [,] = [[0; 10; 20; 30]
                     [1; 11; 21; 31]
                     [2; 12; 22; 32]]
val it : unit = ()
```

length1: `'T [,] -> int`. Returns the length of an array in the first dimension.

Listing 11.94: `Array2D.length1`

```
> let arr = Array2D.create 2 3 0.0 in Array2D.length1 arr;;
val it : int = 2
```

length2: `'T [,] -> int`. Returns the length of an array in the second dimension.

Listing 11.95: `Array2D.forall length2`

```
> let arr = Array2D.create 2 3 0.0 in Array2D.length2 arr;;
val it : int = 3
```

map: `('T -> 'U) -> 'T [,] -> 'U [,]`. Creates a new array whose elements are the results of applying the given function to each of the elements of the array.

Listing 11.96: `Array2D.map`

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j)
- let square x = x*x in Array2D.map square 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]]
```

set: `'T [,] -> int -> int -> 'T -> unit`. Sets the value of an element in an array. You can also use the syntax `array.[index1,index2] <- value`.

Listing 11.97: `Array2D.set`

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j)
- Array2D.set arr 1 2 100
- printfn "%A" arr;;
[[0; 10; 20; 30]
 [1; 11; 100; 31]
 [2; 12; 22; 32]]
val arr : int [,] = [[0; 10; 20; 30]
                     [1; 11; 100; 31]
                     [2; 12; 22; 32]]
val it : unit = ()
```

12 | 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

· Imperative programming
· statements
· states

· mutable values

mutable values
: Mutable values are holders of state, they may change over time, and thus have a dynamic scope.

· procedures

Procedures
: Procedures are functions that returns "`()`", instead of functions that transform data. They are the embodiment of side-effects.

· 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.

· `for`
· `while`

Loops
: The `for`- and `while`-loops typically uses an iteration value to update some state, e.g., `for`-loops are often used to iterate through a list and summarize its content.

In contrast, mono state or stateless programs as *functional programming* can be seen as a subset of imperative programming and is discussed in Chapter 17. *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 Chapter 21.

· functional programming
· Object oriented programming

Imperative programs are like Turing machines, a theoretical machine introduced by Alan Turing in 1936 [10]. Almost all computer hardware is designed for *machine code*, which is a common term used for many low-level computer programming language, and almost all machine languages follow the imperative programming paradigm.

· machine code

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 double size
5. Shape dough into a loaf
6. Let the loaf rise until double size

7. Bake in 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 size of the dough, 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.

12.1 Imperative design

Programming is the act of solving a problem by writing a program to be executed on a computer. And imperative programming focusses 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 2, 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. It is also useful to start in a coarse to fine manner.
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 don't 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 and 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 on 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.

13 Recursion

Recursion is a central concept in F# and used to control flow in loops without the `for` and `while` constructions. Figure 13.1 illustrates the concept of an infinite loop with recursion.

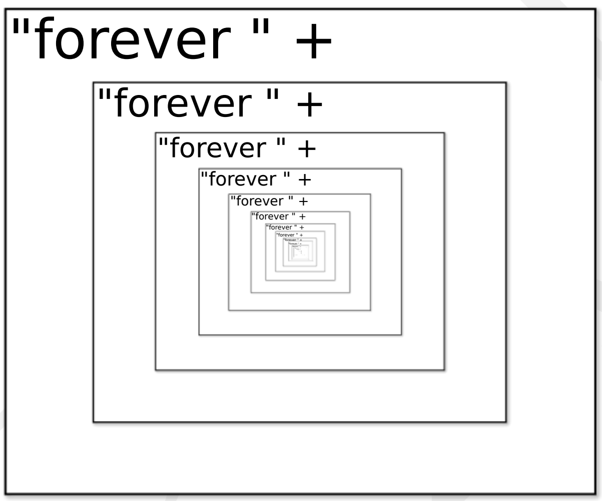


Figure 13.1: An infinitely long string of "forever forever forever...", conceptually calculated by `let rec forever () = "fsharp " + (forever ()).`

13.1 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: · recursive function

Listing 13.1 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. · `rec` · `and`

An example of a recursive function that counts from 1 to 10 similarly to Listing 8.5 is given in Listing 13.2.

**Listing 13.2 countRecursive.fsx:
Counting to 10 using recursion.**

```
let rec prt a b =
  if a > b then
    printf "\n"
  else
    printf "%d " a
    prt (a + 1) b

prt 1 10
```

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

Here the `prt` function calls itself repeatedly, such that the first call is `prt 1 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 Figure 13.2. The old values are no longer accessible as indicated by subscript in the figure. E.g., in $\texttt{prt}_3$ the scope has access to $\texttt{a}_3$ but not $\texttt{a}_2$ and $\texttt{a}_1$. Thus, in this program, 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 13.3 Recursive functions consists of a stopping criterium, a stopping expression, and a recursive step.

```
let rec f a =
  if <stopping condition>
  then <stopping step>
  else <recursion step>
```

The *`match`* – *`with`* are also very common conditional structures. In Listing 13.2 `a > b` is the *stopping condition*, `printfn "\n"` is *stopping step*, and `printfn "\%d " a; prt (a + 1) b` is the *recursion step*.

· `match`
· `with`
· stopping condition
· stopping step
· recursion step

13.2 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. Fibonacci's 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. A direct implementation of this is given in Listing 8.7.

```
$ 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 13.2: Illustration of the recursion used to write the sequence "1 2 3 ... 10" in line 8 in Listing 13.2. Each frame corresponds to a call to `prt`, where new values overshadow old. All return `unit`.

Listing 13.4 fibRecursive.fsx:
The n'th Fibonacci number using recursive.

```
let rec fib n =
  if n < 1 then
    0
  elif n = 1 then
    1
  else
    fib (n - 1) + fib (n - 2)

for i = 0 to 10 do
  printfn "fib(%d) = %d" i (fib i)
```

```
$ fsharpc --nologo fibRecursive.fsx && mono fibRecursive.exe
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
```

Here we extended the sequence to $0, 1, 1, 2, 3, 5, \ldots$ and starting sequence $0, 1$ allowing us to define all $\text{fib}(n) = 0$, $n < 1$. Thus, our function is defined for all integers, and the irrelevant negative arguments fails gracefully by returning 0. This is a general advice: **make functions that fails gracefully.**

Advice

A visualization of the calls and the scopes created by `fibRecursive` is shown in Figure 13.3. The figure illustrates that each recursive step results in two calls to the function, thus creating two new scopes. And it gets worse. Figure 13.4 illustrates the tree of calls for `fib 5`. Thus a call to the function `fib` generates a tree of calls that is five levels deep and has `fib(5)` number of nodes. In general for the program in Listing 13.4, 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 $\alpha \geq \frac{1+\sqrt{5}}{2} \sim 1.6$[1]. 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 calls are identical. E.g., in Figure 13.4 `fib 1` is called five times. Before we examine a faster algorithm, we first need to discuss how F# executes function calls.

When a function is called, then memory is dynamically allocated internally for the function on what is known as the *call stack*. Stacks are used for many things in programming, but typically the call stack is considered special, since it is almost always implicitly part of any program execution. Hence, it is often just referred to as *The Stack*. When a function is called, a new *stack frame* is stacked (pushed) on the call stack including its arguments, local storage such as mutable values, and where execution should return to, when the function is finished. When the function finishes, the stack frame is unstacked (popped) and in its stead the return value of the function is stacked. This return value is then unstacked and used by the caller. After unstacking the return value, the call stack is identical to its state prior to the call. Figure 13.5 shows snapshots of the call stack, when calling `fib 5` in Listing 13.4. The call first 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

· call stack
· The Stack
· stack frame

[1]Jon: `https://math.stackexchange.com/questions/674533/prove-upper-bound-big-o-for-fibonaccis-sequence`

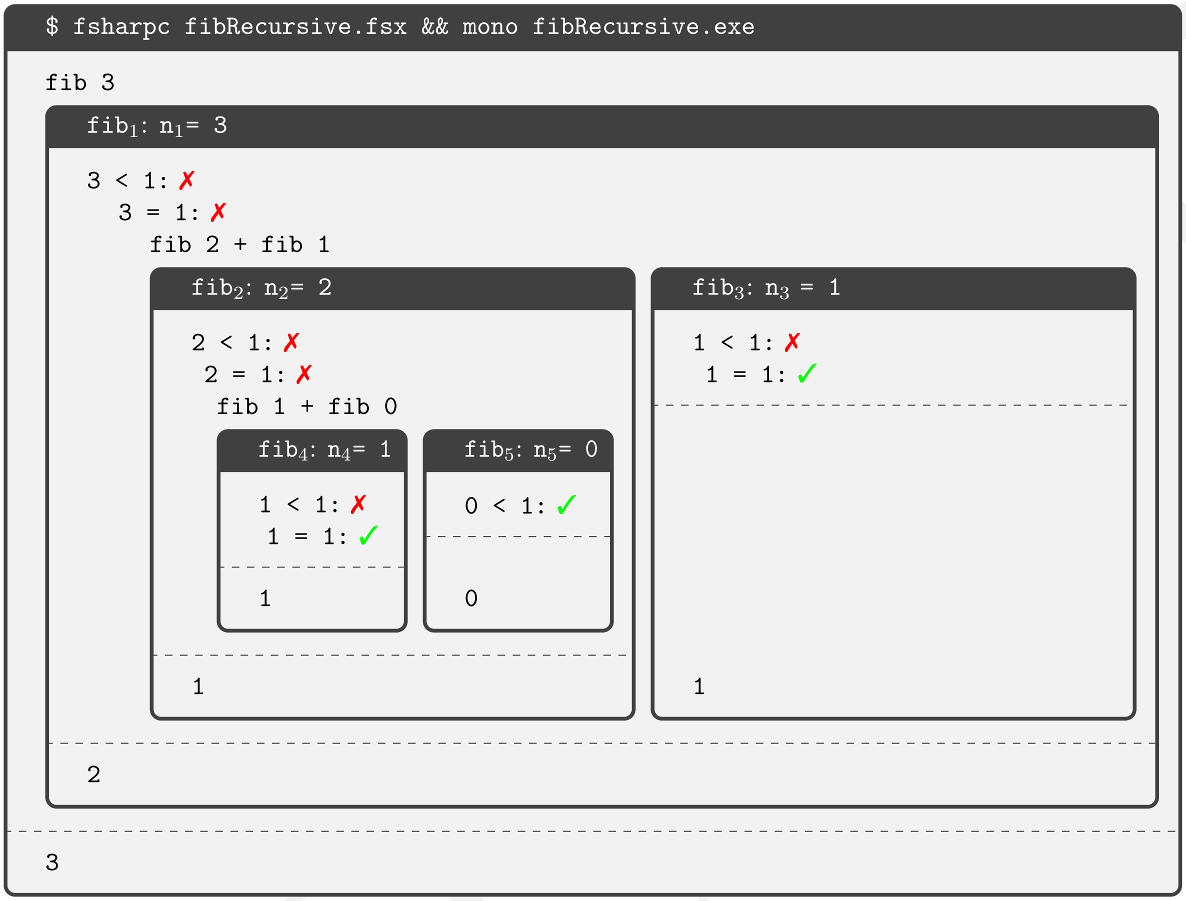


Figure 13.3: Illustration of the recursion used to write the sequence “1 2 3 ... 10” in line 8 in Listing 13.2. Each frame corresponds to a call to `fib`, where new values overshadow old.

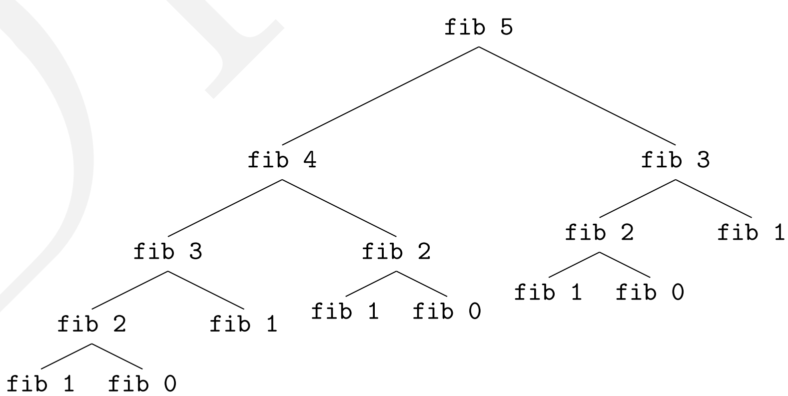


Figure 13.4: The function calls involved in calling `fib 5`.

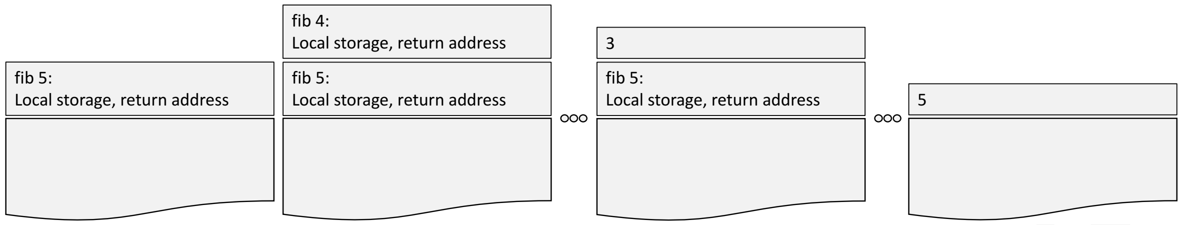


Figure 13.5: A call to `fib 5` in Listing 13.4 starts a sequence of function calls and stack frames on the call stack.

of `fib` is executed, which includes calling `fib 4` and `fib 3` in turn. The call to `fib 4` stacks a frame onto the call stack, and its body is executed. Once execution is returned from the call to `fib 4`, the result of the function is on top of the stack. It is unstacked, saved and the call to `fib 3` is treated equally. When the end of `fib 5` is reached, its frame is unstacked, and its result is stacked. In this way, the call stack is returned to its original state except for the result of the function, and execution is returned to the point right after the original call to `fib 5`. Thus, for Listing 13.4 $\mathcal{O}\left(\alpha^n\right),\ \alpha = \frac{1+\sqrt{5}}{2}$ stacking operations are performed for a call to `fib n`. The $\mathcal{O}\left(f(n)\right)$ 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 $n \geq n_0$, $|g(n)| \leq M|f(n)|$.[2] As indicated by the tree in Figure 13.4, the call tree is maximally n high, which corresponds to a maximum of n additional stack frames as compared to the starting point.

· Landau symbol

The implementation of Fibonacci's sequence in Listing 13.4 can be improved to run faster and use less memory. One such algorithm is given in Listing 13.5

Listing 13.5 fibRecursiveAlt.fsx:
A fast, recursive implementation of Fibonacci's numbers. Compare with Listing 13.4.

```
let fib n =
  let rec fibPair n pair =
    if n < 2 then pair
    else fibPair (n - 1) (snd pair, fst pair + snd pair)
  if n < 1 then 0
  elif n = 1 then 1
  else fibPair n (0, 1) |> snd

printfn "fib(10) = %d" (fib 10)
```

```
$ fsharpc --nologo fibRecursiveAlt.fsx && mono fibRecursiveAlt.exe
fib(10) = 55
```

Calculating the 45th Fibonacci number a Macbook Pro, with a 2.9 Ghz Intel Core i5 using Listing 13.4 takes about 11.2s, while using Listing 13.5 is about 224 times faster and only takes 0.050s. The reason is that `fib` in Listing 13.5 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 `helper` transforms the pair `(a,b)` to `(b,a+b)` such that, e.g., the 4th and 5th pair `(3,5)` is transformed into the 5th and the 6th pair `(5,8)` in the sequence. What complicates the algorithm is that besides the transformation, we must keep track of when to stop, which here is done using a counter variable, that is recursively reduced by 1 until our stopping criterium.

Listing 13.5 also uses much less memory than Listing 13.4, since its recursive call is the last expression

[2] Jon: **Introduction of Landau notation needs to be moved earlier, since it used in Collections chapter.**

in the function, and since the return value of two recursive calls to `helper` is the same as the return value of the last. In fact, the return value of any number of recursive calls to `helper` 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 `helper` calls itself, then its frame is no longer needed, and may be replaced by the new `helper` with the slight modification, that the return point should be to `fib` and not the end of the previous `helper`. Once the recursion reaches the stopping criteria, then instead of popping a long list of calls of `helper` 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 is replaced by one. Hence, **prefer tail-recursion whenever possible.**

· tail-recursion

Advice

13.3 Mutual 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)`:

· mutually recursive
· let
· rec
· and

Listing 13.6 mutuallyRecursive.fsx:
Using mutual recursion to implement even and odd functions.

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

let w = 5;
printfn "%*s %*s %*s" w "i" w "even" w "odd"
for i = 1 to w do
  printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
$ fsharpc --nologo mutuallyRecursive.fsx && mono mutuallyRecursive.exe
    i  even   odd
    1 false  true
    2  true false
    3 false  true
    4  true false
    5 false  true
```

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.,

Listing 13.7 mutuallyRecursiveAlt.fsx:
Mutual recursion without the and keyword needs a helper function.

```
let rec evenHelper (notEven: int -> bool) x =
  if x = 0 then true
  else notEven (x - 1)

let rec odd x =
  if x = 0 then false
  else evenHelper odd (x - 1);;

let even x = evenHelper odd x

let w = 5;
printfn "%*s %*s %*s" w "i" w "Even" w "Odd"
for i = 1 to w do
  printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
$ fsharpc --nologo mutuallyRecursiveAlt.fsx
$ mono mutuallyRecursiveAlt.exe
    i  Even   Odd
    1 false  true
    2  true false
    3 false  true
    4  true false
    5 false  true
```

But, Listing 13.6 is clearly to be preferred over Listing 13.7.

In the 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:

Listing 13.8 parity.fsx:
A better way to test for parity without recursion.

```
let even x = (x % 2 = 0)
let odd x = not (even x)
```

which is to be preferred anytime as the solution to the problem. [3]

[3] Jon: **Here it would be nice to have an intermezzo, giving examples of how to write a recursive program by thinking the problem has been solved.**

14 | 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 useful for simple programming tasks, and everything that can be programmed can be accomplished using these types. However, larger programs are often easier to read and write when using more complicated type structures. In this chapter, we will discuss type abbreviations, enumerated types, discriminated unions, records and structs. Class types are discussed in depth in Chapter 20.

· primitive types

14.1 Type abbreviations

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

· type abbreviation
· type aliasing

Listing 14.1 Syntax for type abbreviation.

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

where the identifier is a new name, and the type-name is an existing or a compound of existing types. E.g., in Listing 14.2 several type abbreviations are defined.

Listing 14.2 typeAbbreviation.fsx:
Defining 3 type abbreviations, two of which are compound types.

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

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

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

Here we define the abbreviations `size`, `position`, `person`, and `intToFloat`, and later make bindings enforcing the usage of these abbreviations.

Type abbreviations are useful 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 only can 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.

14.2 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:

· enumerations
· enums

Listing 14.3 Syntax for enumerations.

```
type <ident> =
  [ | ] <ident> = <integerOrChar>
  | <ident> = <integerOrChar>
  | <ident> = <integerOrChar>
  ...
```

An example of using enumerations is given in Listing 14.4.

Listing 14.4 enum.fsx:
An enum type acts as a typed alias to a set of integers or chars.

```
type medal =
  Gold = 0
  | Silver = 1
  | Bronze = 2

let aMedal = medal.Gold
printfn "%A has value %d" aMedal (int aMedal)
```

```
$ fsharpc --nologo enum.fsx && mono enum.exe
Gold has value 0
```

In the 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 requires 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.

14.3 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 14.5 Syntax for type abbreviation.

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

Discriminated unions are reference types, i.e., their content is stored on *The Heap*, see Section 6.7 for a discussion on reference types. Since they are immutable, there is no risk of side-effects. As reference types, when used as arguments to and returned from a function, then only a reference is passed. 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. However, there is a slight overhead, since working with the content of reference types is indirect through their reference. Discriminated unions can also be represented as structures using the [<Struct>] attribute, in which case they are value types. See Section 14.5 for a discussion on structs.

· The Heap

An example just using the named cases but no further specification of types is given in Listing 14.6.

Listing 14.6 discriminatedUnions.fsx:
A discriminated union of medals. Compare with Listing 14.4.

```
type medal =
  Gold
  | Silver
  | Bronze

let aMedal = medal.Gold
printfn "%A" aMedal
```

```
$ fsharpc --nologo discriminatedUnions.fsx
$ mono discriminatedUnions.exe
Gold
```

Here we define a discriminated union as three named cases signifying three different types of medals. Comparing with the enumerated type in Listing 14.4, we see that the only difference is that the cases of the discriminated unions have no value. A commonly used discriminated union is the *option type*, see Section 18.2 for more detail.

· option type

Discriminated unions may also be used to store data. Where the names in enumerated types are aliases of single values, the names in discriminated unions can hold any value specified at the time of creation. An example is given in Listing 14.7.

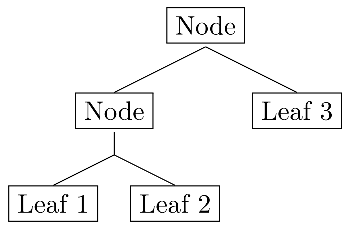


Figure 14.1: The tree with 3 leaves.

Listing 14.7 discriminatedUnionsOf.fsx:
A discriminated union using explicit subtypes.

```
type vector =
  Vec2D of float * float
  | Vec3D of x : float * y : float * z : float

let v2 = Vec2D (1.0, -1.2)
let v3 = Vec3D (x = 1.0, z = -1.2, y = 0.9)
printfn "%A and %A" v2 v3
```

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

In this 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. As can be seen, the arguments may be given field names, and if they are, then the names may be used when creating values of this type. As also demonstrated, the field names can be used to specify the field values in arbitrary order. However, values for all fields must be given.

Discriminated unions can be defined recursively. This feature is demonstrated in Listing 14.8.

Listing 14.8 discriminatedUnionTree.fsx:
A discriminated union modelling binary trees.

```
type Tree =
  Leaf of int
  | Node of Tree * Tree

let one = Leaf 1
let two = Leaf 2
let three = Leaf 3
let tree = Node (Node (one, two), three)
printfn "%A" tree
```

```
$ fsharpc --nologo discriminatedUnionTree.fsx
$ mono discriminatedUnionTree.exe
Node (Node (Leaf 1,Leaf 2),Leaf 3)
```

In this example we define a tree as depicted in Figure 14.1.

Pattern matching must be used in order to define functions on values of discriminated union. E.g., in

Listing 14.9 we define a function that traverses a tree and prints the content of the nodes.[1]

Listing 14.9 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
```

Discriminated unions are very powerful and can often be used instead of class hierarchies. Class hierarchies are discussed in Section 20.10.

14.4 Records

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

Listing 14.10 Syntax for defining record types.

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

Records are collections of named variables and values of varied type. They are reference types, and thus their content is stored on *The Heap*, see Section 6.7 for a discussion on reference types. Records can also be *struct records* using the [<Struct>] attribute, in which case they are value types. See Section 14.5 for a discussion on structs. An example of using records is given in Listing 14.11. The values of individual members of a record is obtained using the "." notation

· The Heap
· struct records
· .

[1] Jon: **Example uses pattern matching, which has yet to be introduced.**

Listing 14.11 records.fsx:
A record is defined for holding information about a person.

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

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

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

The examples illustrates a how record type is defined to store varied data about a person, and how a value is created by a record expression defining its field values.

If two record types are defined with the same label set, then the later dominates the former, and the compiler will at a binding infer that later. This is demonstrated in Listing 14.12.

Listing 14.12 recordsDominance.fsx:
Redefined types dominates old record types, but earlier definitions are still accessible using explicit or implicit specification for bindings.

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

let lecturer = {name = "Jon"; age = 50; height = 1.75}
printfn "%A : %A" lecturer (lecturer.GetType())
let author : person = {name = "Jon"; age = 50; height = 1.75}
printfn "%A : %A" author (author.GetType())
let father = {person.name = "Jon"; age = 50; height = 1.75}
printfn "%A : %A" author (author.GetType())
```

```
$ fsharpc --nologo recordsDominance.fsx && mono recordsDominance.exe
{name = "Jon";
 age = 50;
 height = 1.75;} : RecordsDominance+teacher
{name = "Jon";
 age = 50;
 height = 1.75;} : RecordsDominance+person
{name = "Jon";
 age = 50;
 height = 1.75;} : RecordsDominance+person
```

In the example, two identical record types are defined, and we use the builtin `GetType()` method to inspect the type of bindings. We see that `lecturer` is of `RecordsDominance+teacher` type, since `teacher` dominates the identical `author` 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 therefore of `RecordsDominance+author` type. The built-in `GetType()` method is inherited from the base class for all types, see Chapter 20 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, e.g., `{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 Listing 14.13. · with

Listing 14.13 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
```

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

Here `age` is defined as a mutable value, and can be changed using the usual "`<-`" assignment operator. The example demonstrates two different ways to create records. Note that when the mutable value `author.age` is changed in line 10, then `authorAlias` also changes, since it is an alias of `author`, but neither `authorCopy` nor `authorCopyAlt` changes, since they are copies. As illustrated, copying using `with` allows for easy copying and partial updates of another record value.

14.5 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, then they are best used for small amount of data. The syntax for defining struct types are: · structures · structs

Listing 14.14 Syntax for type abbreviation.

```
[ <attributes> ]
[<Struct>]
type <ident> =
  val [ mutable ] <label1> : <type1>
  val [ mutable ] <label2> : <type2>
  ...
  [new (<arg1>, <arg2>, ...) = {<label1> = <arg1>; <label1> = <arg2>; ...}
  [new (<arg1>, <arg2>, ...) = {<label1> = <arg1>; <label1> = <arg2>; ...}
  ...
```

The syntax makes use of the `val` and `new` keywords. Keyword `val` like `let` 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` nor `do` bindings are allowed in structure definitions. Fields are accessed using the "`.`" notation. An example is given in Listing 14.15.

· `val`
·
· `new`
· constructor
· `.`

Listing 14.15 struct.fsx:
Defining a struct type and creating a value of it.

```
[<Struct>]
type position =
  val x : float
  val y : float
  new (a : float, b : float) = {x = a; y = b}

let p = position (3.0, 4.2)
printfn "%A: x = %A, y = %A" p p.x p.y
```

```
$ fsharpc --nologo struct.fsx && mono struct.exe
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 Listing 14.16.

· overload
· override
· `ToString()`

Listing 14.16 structOverloadNOverride.fsx:
Overloading the new constructor and overriding the default ToString() function.

```
[<Struct>]
type position =
  val x : float
  val y : float
  new (a : float, b : float) = {x = a; y = b}
  new (a : int, b : int) = {x = float a; y = float b}
  override this.ToString() =
    "(" + (string this.x) + ", " + (string this.y) + ")"

let pFloat = position (3.0, 4.2)
let pInt = position (3, 4)
printfn "%A and %A" pFloat pInt
```

```
$ fsharpc --nologo structOverloadNOverride.fsx
$ mono structOverloadNOverride.exe
(3, 4.2) and (3, 4)
```

We defer further discussion of these concepts to Chapter 20.

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.

14.6 Variable types

An advanced topic in F# is *variable types*. There are three different versions of variable types in F#: *runtime resolved*, which has the syntax '<ident>, *anonymous*, which are written as "_", and *statically resolved*, which have the syntax ^<ident>. Variable types are particularly useful for functions that work for many types. An example of a generic function and its use is given in Listing 14.17.

· variable types
· runtime resolved variable type
· anonymous variable type
· _
· statically resolved variable type

Listing 14.17 variableType.fsx:
A function apply with runtime resolved types.

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

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

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

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.

The example in Listing 14.17 illustrates a very complicated way to add two numbers. And the "+" operator works for both types out of the box, so why not something simpler like relying on the F# type inference system by not explicitly specifying types as attempted in Listing 14.18.

Listing 14.18 variableTypeError.fsx:
Even though the "+" operator is defined for both integers and floats, the type inference is static and infers `plus : int -> int`.

```
let plus x y = x + y

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

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

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

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

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 Listing 14.19. · `inline`

Listing 14.19 variableTypeInline.fsx:
The keyword `inline` forces static and independent inference each place the function is used. Compare to the error case in Listing 14.18.

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

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

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

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 Listing 14.20.

Listing 14.20 compiletimeVariableType.fsx:
Explicitely spelling out of the statically resolved type variables from Listing 14.18.

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

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

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

The example in Listing 14.20 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.[2] 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 which where the type can be inferred at compile time. See Chapter 20 for details on member functions.

· type constraints
· `when`

The use of `inline` 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 type. E.g., the "+" operator has type `( + ) : ^T1 -> ^T2 -> ^T3 (requires ^T1 with static member (+) and ^T2 with static member (+))`.

[2]Jon: **Should I extend on type constraints? Perhaps it is better left for a specialize chapter on generic functions.**

15 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 Section 6.1, its extension with pattern matching is given as,

· let

Listing 15.1 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 Listing 15.2.

Listing 15.2 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
```

Here 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 alternative to `if` – `then` – `else` expressions particularly when parsing input for a function. Consider the example in Listing 15.3.

Listing 15.3 switch.fsx:
Using `if` – `then` – `else` to print discriminated unions.

```
type Medal = Gold | Silver | Bronze
let statement (m : Medal) : string =
  if m = Gold then "You won"
  elif m = Silver then "You almost won"
  else "Maybe you can win next time"

let m = Silver
printfn "%A : %s" m (statement m)
```

```
$ fsharpc --nologo switch.fsx && mono switch.exe
Silver : You almost won
```

In the example, is a discriminated union and a function defined. The funciton converts each case to a supporting statement using an `if`-expression. The same can be done with the *`match`* – *`with`* expression and patterns as is demonstrated in Listing 15.4.

· `match`
· `with`

Listing 15.4 switchPattern.fsx:
Using `match` – `with` to print discriminated unions.

```
type Medal = Gold | Silver | Bronze
let statement (m : Medal) : string =
  match m with
    Gold -> "You won"
    | Silver -> "You almost won"
    | _ -> "Maybe you can win next time"

let m = Silver
printfn "%A : %s" m (statement m)
```

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

Here we used a pattern for the discriminated union cases and a wildcard pattern as default. The lightweight syntax for `match`-expressions is,

Listing 15.5 Syntax for `match`-expressions.

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

where `<inputExpr>` is the *input pattern* to find matches of, `<pat>` is a pattern to match with, `<guardExpr>` is an optional guard expression, and `<caseExpr>` is the resulting expression. Each set starting with `<pat>` is called a case. 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# report an error, when not the complete domain of the input pattern is covered by cases in `match`-expressions.

· input pattern

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

Listing 15.6 Syntax for `for`-expressions with pattern matching.

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

Typically, `<sourceExpr>` is a list or an array. An example is given in Listing 15.7.

Listing 15.7 forPattern.fsx:
Patterns may be used in `for`-loops.

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

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

The 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.** Advice

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

Listing 15.8 Syntax for anonymous functions with pattern matching.

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

This is an extension of the syntax discussed in Section 6.2. A typical use for patterns in *`fun`*-expressions is shown in Listing 15.9. · `fun`

Listing 15.9 funPattern.fsx:
Patterns may be used in `fun`-expressions.

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

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

Here we use an anonymous function expression and bind it to `f`. The expression has one argument of any type, which it ignores using the wildcard pattern. Some limitations apply to the patterns allowed in `fun`-expressions.[1] 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. · mockup functions

Patterns are also used in exceptions to be discussed in Section 18.1, and in conjunction with the

[1] Jon: **Remove or elaborate.**

`function`-keyword, a keyword we discourage in this book. We will now demonstrate a list of important patterns in F#.

15.1 Wildcard pattern

A *wildcard pattern* is denoted "`_`" and matches anything, see e.g., Listing 15.10.

· wildcard pattern
· `_`

Listing 15.10 wildcardPattern.fsx:
Constant patterns matches to constants.

```
let whatEver (x : int) : string =
  match x with
    _ -> "If you say so"

printfn "%s" (whatEver 42)
```

```
$ fsharpc --nologo wildcardPattern.fsx && mono wildcardPattern.exe
If you say so
```

In this 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.

15.2 Constant and literal patterns

A *constant pattern* matches any input pattern with constants, see e.g., Listing 15.11.

· constant pattern

Listing 15.11 constPattern.fsx:
Constant patterns matches to constants.

```
type Medal = Gold | Silver | Bronze
let intToMedal (x : int) : Medal =
  match x with
    0 -> Gold
    | 1 -> Silver
    | _ -> Bronze

printfn "%A" (intToMedal 0)
```

```
$ fsharpc --nologo constPattern.fsx && mono constPattern.exe
Gold
```

In this example, the input pattern are queried for a match with `0` and `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. Notice, matching is performed in a lazy manner and stops for 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 matches the input.

Constants can also be pre-bound by the `[<Literal>]` attribute for value-bindings. This is demonstrated in Listing 15.12.

Listing 15.12 literalPattern.fsx:
A variant of constant patterns are literal patterns.

```
[<Literal>]
let TheAnswer = 42
let whatIsTheQuestion (x : int) : string =
  match x with
    TheAnswer -> "We will need to build a bigger machine..."
    | _ -> "Don't know that either"

printfn "%A" (whatIsTheQuestion 42)
```

```
$ fsharpc --nologo literalPattern.fsx && mono literalPattern.exe
"We will need to build a bigger machine..."
```

The attributed 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.

15.3 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 patters are also allowed together with records and arrays. This is demonstrated in Listing 15.13.

· variable pattern

Listing 15.13 variablePattern.fsx:
Variable patterns are useful for extracting and naming fields etc.

```
let (name, age) = ("Jon", 50)
let getAgeString (age : int) : string =
  match age with
    0 -> "newborn"
    | 1 -> "1 year old"
    | n -> (string n) + " years old"

printfn "%s is %s" name (getAgeString age)
```

```
$ fsharpc --nologo variablePattern.fsx && mono variablePattern.exe
Jon is 50 years old
```

In this example, the use of the value identifier `n` has the function of a 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 Section 15.4.

15.4 Guards

A *guard* is a pattern used together with match-expressions including the *when*-keyword, as shown in Listing 15.5.

· guard
· when

Listing 15.14 guardPattern.fsx:
Guard expressions can be used with other patterns to restrict matches.

```
let getAgeString (age : int) : string =
  match age with
    n when n < 1 -> "infant"
    | n when n < 13 -> "child"
    | n when n < 20 -> "teen"
    | _ -> "adult"

printfn "A person aged %d is a/an %s" 50 (getAgeString 50)
```

```
$ fsharpc --nologo guardPattern.fsx && mono guardPattern.exe
A person aged 50 is a/an adult
```

Here guards are used to iteratively carve out subset of integers to assign 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.

15.5 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 Listing 15.15

· list pattern
· ::
· []

Listing 15.15 listPattern.fsx:
Recursively parsing a list using list patterns.

```
let rec sumList (lst : int list) : int =
  match lst with
    n :: rest -> n + (sumList rest)
    | [] -> 0

let rec sumThree (lst : int list) : int =
  match lst with
    [a; b; c] -> a + b + c
    | _ -> sumList lst

let aList = [1; 2; 3]
printfn "The sum of %A is %d, %d" aList (sumList aList) (sumThree aList)
```

```
$ fsharpc --nologo listPattern.fsx && mono listPattern.exe
The sum of [1; 2; 3] is 6, 6
```

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.

15.6 Array, record, and discriminated union patterns

Array, *record*, and *discriminated union patterns* are direct extensions on constant, variable, and wildcard patterns. Listing 15.16 gives examples of array patterns.

· array pattern
· record pattern
· discriminated union patterns

Listing 15.16 arrayPattern.fsx:
Using variable patterns to match on size and content of arrays.

```
let arrayToString (x : int []) : string =
  match x with
    [|1;_;_|] -> "3 elements, first of is a one"
    | [|x;1;_|] -> "3 elements, first is " + (string x) + "Second is one"
    | x -> "A general array"

printfn "%s" (arrayToString [|1; 1; 1|])
printfn "%s" (arrayToString [|3; 1; 1|])
printfn "%s" (arrayToString [|1|])
```

```
$ fsharpc --nologo arrayPattern.fsx && mono arrayPattern.exe
3 elements, first of is a one
3 elements, first is 3Second is one
A general array
```

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 pattern, we use the field names to specify matching criteria. This is demonstrated in Listing 15.17.

Listing 15.17 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
```

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 needs 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 match as constants, variables, or wildcards. A demonstration is given in Listing 15.18.

Listing 15.18 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)
```

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 which case "`;`" is used to delimit the fields in the match instead of "`,`".

15.7 Disjunctive and conjunctive patterns

Patterns may be combined disjunctively using the "|" lexeme and conjunctively using the "&" lexeme. *Disjunctive patterns* combine as fall-through as illustrated in Listing 15.19.

· |
· &
· disjunctive pattern

Listing 15.19 disjunctivePattern.fsx:
Patterns can be combined logically as 'or' syntax structures.

```
let wovel (c : char) : bool =
  match c with
    'a' | 'e' | 'i' | 'o' | 'u' | 'y' -> true
    | _ -> false

String.iter (fun c -> printf "%A " (wovel c)) "abcdefg"
```

```
$ fsharpc --nologo disjunctivePattern.fsx && mono disjunctivePattern.exe
true false false false true false false
```

Here one or more cases must match for the final case expression, and thus, any vowel results in the same case expression `true`. All else is matched with the wildcard pattern.

For *conjunctive patterns* all patterns must match, which is illustrated in Listing 15.20.

· conjunctive patterns

Listing 15.20 conjunctivePattern.fsx:
Patterns can be combined logically as 'or' syntax structures.

```
let is11 (v : int * int) : bool =
  match v with
    (1,_) & (_,1) -> true
    | _ -> false

printfn "%A" (List.map is11 [(0,0); (0,1); (1,0); (1,1)])
```

```
$ fsharpc --nologo conjunctivePattern.fsx && mono conjunctivePattern.exe
[false; false; false; true]
```

In 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 useful together with active patterns, to be discussed below.

15.8 Active Pattern

The concept of patterns is extendable to functions. Such functions are called *active patterns*, and active patterns comes 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,

· active patterns

Listing 15.21 Syntax for binding active patterns to expressions.

```
let (|<caseName>|[_| ]) [ <arg> [<arg> ... ]] <inputArgument> = <expr>
let (|<caseName>|<caseName>|...|<caseName>|) <inputArgumetn> = <expr>
```

When using the (|<caseName>|_|]) variants, then the active pattern function must return an option type. The multi-case variant (|<caseName>|<caseName>|...|<caseName>|) 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 how the various patterns are used by example.

The single case, (|<caseName>|]) matches all, and is useful for extracting information from complex types, as demonstrated in Listing 15.22.

Listing 15.22 activePattern.fsx:
Single case active pattern for deconstructing complex types.

```
1  type vec = {x : float; y : float}
2  let (|Cartesian|) (v : vec) = (v.x, v.y)
3  let (|Polar|) (v : vec) = (sqrt(v.x*v.x + v.y * v.y), atan2 v.y v.x)
4  let printCartesian (p : vec) : unit =
5      match p with
6        Cartesian (x, y) -> printfn "%A:\n Cartesian (%A, %A)" p x y
7  let printPolar (p : vec) : unit =
8      match p with
9        Polar (a, d) -> printfn "%A:\n Cartesian (%A, %A)" p a d
10
11 let v = {x = 2.0; y = 3.0}
12 printCartesian v
13 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;}:
 Cartesian (3.605551275, 0.9827937232)
```

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 2 and 3, the argument is the input expression match <inputExpr> with ..., see Listing 15.5. However, the argument for the cases in line 6 and 9 are names bound to the output of the active pattern function.

Both Cartesian and Polar matches a vector record, but they dismantle the contents differently. For an alternative solution using Class types, see Section 20.1.

More complicated behavior is obtainable by supplying additional arguments to the single case. This is demonstrated in Listing 15.23.

Listing 15.23 activeArgumentsPattern.fsx:
All but the multi-case active pattern may take additional arguments.

```
1  type vec = {x : float; y : float}
2  let (|Polar|) (o : vec) (v : vec) =
3    let x = v.x - o.x
4    let y = v.y - o.y
5    (sqrt(x*x + y * y), atan2 y x)
6  let printPolar (o : vec) (p : vec) : unit =
7      match p with
8        Polar o (a, d) -> printfn "%A:\n Cartesian (%A, %A)" p a d
9
10 let v = {x = 2.0; y = 3.0}
11 let offset = {x = 1.0; y = 1.0}
12 printPolar offset v
```

```
1  $ fsharpc --nologo activeArgumentsPattern.fsx
2  $ mono activeArgumentsPattern.exe
3  {x = 2.0;
4   y = 3.0;}:
5   Cartesian (2.236067977, 1.107148718)
```

Here we supply an offset, which should be subtracted prior to calculating lengths and angles. Notice in line 8, that the 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 Listing 15.24.

· partial pattern functions

Listing 15.24 activeOptionPattern.fsx:
Option type active patterns mismatches on `None` results.

```
1  let (|Div|_|) (e,d) = if d <> 0.0 then Some (e/d) else None
2
3  let safeDiv (p : float * float) =
4      match p with
5        | (0.0, 0.0) -> printfn "Div %A = undefined" p
6        | Div res -> printfn "Div %A = %A" p res
7        | _ -> printfn "Div %A = infinity" p
8
9  List.iter safeDiv [(1.0,1.0); (0.0,1.0); (1.0,0.0); (0.0,0.0)]
```

```
1  $ fsharpc --nologo activeOptionPattern.fsx
2  $ mono activeOptionPattern.exe
3  Div (1.0, 1.0) = 1.0
4  Div (0.0, 1.0) = 0.0
5  Div (1.0, 0.0) = infinity
6  Div (0.0, 0.0) = undefined
```

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 6 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.[2] An example is given in Listing 15.25.

· multicase active patterns

Listing 15.25 activeMultiCasePattern.fsx: Multi-case active patterns have a syntactical structure similar to discriminated unions.

```
1  let (|Gold|Silver|Bronze|) inp =
2    if inp = 0 then Gold
3    elif inp = 1 then Silver
4    else Bronze
5
6  let intToMedal (i : int) =
7      match i with
8        Gold -> printfn "%d: Its gold!" i
9        | Silver -> printfn "%d: Its silver." i
10       | Bronze -> printfn "%d: Its no more than bronze." i
11
12 List.iter intToMedal [0..3]
```

```
$ fsharpc --nologo activeMultiCasePattern.fsx
$ mono activeMultiCasePattern.exe
0: Its gold!
1: Its silver.
2: Its no more than bronze.
3: Its no more than bronze.
```

In this example, we define three cases in line 1. 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 8, 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.

15.9 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 Listing 15.26.

· static type pattern
· :

[2] Jon: **This may be too advanced for this book.**

Listing 15.26 staticTypePattern.fsx:
Static matching on type binds the type of other values by type inference.

```
let rec sum lst =
    match lst with
      (n : int) :: rest -> n + (sum rest)
      | [] -> 0

printfn "The sum is %d" (sum [0..3])
```

```
$ fsharpc --nologo staticTypePattern.fsx && mono staticTypePattern.exe
The sum is 6
```

Here the head of the list `n` in the list pattern is explicitly matched as an integer, 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 runtime, 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 Listing 15.27.

· dynamic type pattern
· `:?`

Listing 15.27 dynamicTypePattern.fsx:
Static matching on type binds the type of other values by type inference.

```
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
```

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 Chapter 20 for more on objects. Dynamic type patterns are often used for analysing exceptions, which is discussed in Section 18.1. While dynamic type patterns are useful, they imply runtime checking, and **it is almost always better to prefer compile time than runtime type checking.**

Advice

16 | 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 Section 6.2, which can be passed to and from functions as any other value. An example of a higher order function is `List.map`, which takes a function and a list and produces a list, demonstrated in Listing 16.1.

· higher order function

· closures

Listing 16.1 higherOrderMap.fsx:
`List.map` is a higher order function, since it takes a function as argument.

```
let inc x = x + 1
let newList = List.map inc [2; 3; 5]
printfn "%A" newList
```

```
$ fsharpc --nologo higherOrderMap.fsx && mono higherOrderMap.exe
[3; 4; 6]
```

Here `List.map` applies the function `inc` to every element of the list. *Anonymous functions* are expressions that results in a function, which need not be named. A typical example is to use anonymous functions as arguments to higher order functions such as `List.map` shown in Listing 16.2.

· anonymous functions

Listing 16.2 higherOrderAnonymous.fsx:
An anonymous function is a higher order function used here as an unnamed argument. Compare with Listing 16.1.

```
let newList = List.map (fun x -> x + 1) [2; 3; 5]
printfn "%A" newList
```

```
$ fsharpc --nologo higherOrderAnonymous.fsx
$ mono higherOrderAnonymous.exe
[3; 4; 6]
```

A very brief version of Listing 16.2 is shown in Listing 16.3.

Listing 16.3 higherOrderAnonymousBrief.fsx:
A compact version of Listing 16.1.

```
printfn "%A" (List.map (fun x -> x + 1) [2; 3; 5])
```

```
$ fsharpc --nologo higherOrderAnonymousBrief.fsx
$ mono higherOrderAnonymousBrief.exe
[3; 4; 6]
```

What was originally three lines in Listing 16.1 as been reduced to a single line in Listing 16.3. While the result is exactly the same, the later is more difficult to read: First the reader has to understand what the anonymous function does on a single argument, then how this applies to the list through `list.Map`, and finally what `printfn` does with the result.

Anonymous functions are also useful as return values of functions as shown in Listing 16.4

Listing 16.4 higherOrderReturn.fsx:
The procedure `inc` returns an increment function. Compare with Listing 16.1.

```
let inc n =
  fun x -> x + n
printfn "%A" (List.map (inc 1) [2; 3; 5])
```

```
$ fsharpc --nologo higherOrderReturn.fsx && mono higherOrderReturn.exe
[3; 4; 6]
```

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 Listing 16.1.

Piping is another example of a set of higher order function: `(<|)`, `(|>)`, `(<||)`, `(||>)`, `(<|||)`, `(|||>)`.[1] 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 Listing 16.5. · piping

Listing 16.5 higherOrderPiping.fsx:
The functional equivalent of the right-to-left piping operator is a higher order function.

```
let inc x = x + 1
let aValue = 2
let anotherValue = (|>) aValue inc
printfn "%d -> %d" aValue anotherValue
```

```
$ fsharpc --nologo higherOrderPiping.fsx && mono higherOrderPiping.exe
2 -> 3
```

Here the piping operator is used to apply the `inc` function to `aValue`. A more elegant way to write

[1] Jon: **Make piping operators go into index.**

this would be `aValue |> inc`, or even just `inc aValue`.

16.1 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 Listing 16.6.

· function composition
· >>
· <<

**Listing 16.6 higherOrderComposition.fsx:
Functions defined as composition of other functions.**

```
let f x = x + 1
let g x = x * x
let h = f >> g
let k = f << g
printfn "%d" (g (f 2))
printfn "%d" (h 2)
printfn "%d" (f (g 2))
printfn "%d" (k 2)
```

```
$ fsharpc --nologo higherOrderComposition.fsx
$ mono higherOrderComposition.exe
9
9
5
5
```

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` and applies the function to the value and produces the value `'b`. In contrast, the 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`.

16.2 Currying

Consider a function `f` of two generic arguments. It's 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

· partial specification

currying in tribute of Haskell Curry[2]. An example is given in Listing 16.7.

Listing 16.7 higherOrderCurrying.fsx:
Currying: defining a function as a partial specification of another.

```
let mul x y = x*y
let timesTwo = mul 2.0
printfn "%g" (mul 5.0 3.0)
printfn "%g" (timesTwo 3.0)
```

```
$ fsharpc --nologo higherOrderCurrying.fsx
$ mono higherOrderCurrying.exe
15
6
```

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 Listing 16.8.

Listing 16.8 higherOrderTuples.fsx:
Partial specification of functions using tuples is less elegant. Compare with Listing 16.7.

```
let mul (x, y) = x*y
let timesTwo y = mul (2.0, y)
printfn "%g" (mul (5.0, 3.0))
printfn "%g" (timesTwo 3.0)
```

```
$ fsharpc --nologo higherOrderTuples.fsx && mono higherOrderTuples.exe
15
6
```

Conversion between multiple and tuple arguments is easily done with higher order functions as demonstrated in Listing 16.9.

Listing 16.9: 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
```

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 Listing 16.8, then `curry mul` has the type `x:'a -> y:'b -> 'c` as can be seen in Listing 16.9, 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

[2]Haskell Curry (1900–1982) was an American mathematician and logician, who also has a programming language named after him: Haskell.

may lead to lower readability of code, and in generally **currying should be used with care and be well documented for proper readability of code.** Advice

17 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 Chapter 21. 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 attempt to rectify this, however, the argument still holds.

The functional programming paradigm can trace its roots to lambda calculus introduced by Alonzo Church in 1936 [1]. Church designed lambda calculus to discuss computability. Some of the forces of the functional programming paradigm is that it is often easier to prove 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 function

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*.

· referential transparency

· higher order function

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.

· first-class citizenship

· recursion

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.

· iter
· map
· reduce
· fold
· foldback

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

· immutable state
· immutable state
· strongly typed

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. The implication is that type errors are caught at compile time instead of at runtime.

Lazy evaluation · lazy evaluation

Due to referential transparency, values can be compute any time up until the point, when 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. · `lazy` · `seq`

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 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 change the value at that location. This has constant computational complexity. While fast, mutable values give dynamic scope, 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 has also 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, and less open for, e.g., hacking. · imperative programming

17.1 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. Better is 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 produce small, well-defined functions, there is a tendency to the usage of 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 outer most scope, while decreasing in size as you move in.

- Few comments. Since functional programming is very concise, there is a tendency, that we as programmers forget to add sufficient comments to the code, since at the time of writing, the meaning may be very clear and well thought through, but 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 family of identifiers like `a`, `a2`, `newA` etc. Better is to use names, that more clearly states the semantic meaning the values, or, if only used as temporary storage, to discard them completely in lieu of piping and function composition. However, the latter most often requires comments describing the transformation being performed.

Thus, a design pattern for functional programs must focus on,

- What is the input data to be processed
- How is the data to be transformed

For large programs, the design principle is often similar to other programming paradigms, which are often visualized graphically as components that take input, interact, and produces 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 structure functionally.

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.

· graphical user interface
· GUI
· console
· terminal
· Windows command line

A.1 The basics

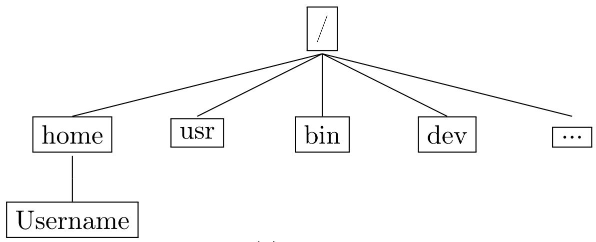
When you open a *directory* or *folder* in your preferred operating system, the directory will have a location in the file system, whether from the console or through the operating system's graphical user interface. The console will almost always be associated with a particular directory or folder in 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, MaxOS X and Windows, but common is that it is a hierarchical structure. This is illustrated in Figure A.1.

· directory
· folder

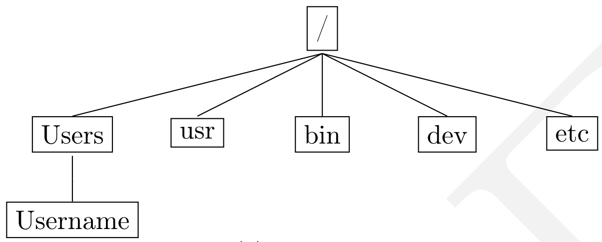
There are many predefined console commands, available in the console, and you can make your own. In the following, we will review the most important commands in the three different operating systems. These are summarized in Table A.1.

A.2 Windows

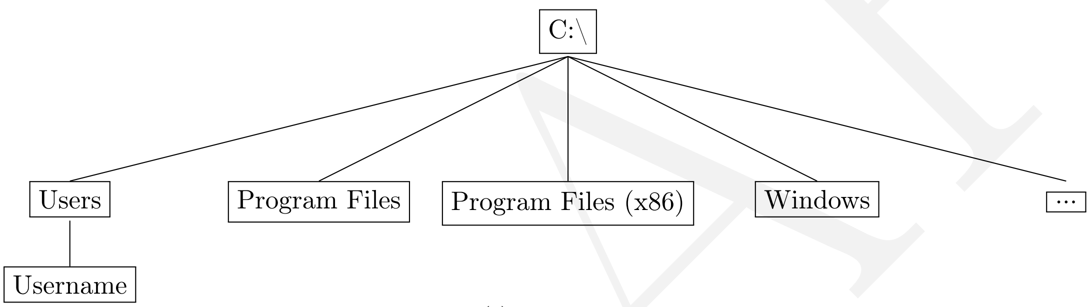
In this section we will discuss the commands summarized in Table A.1. Windows 7 and earlier versions: To open the console, press `Start->Run` in the lower left corner, and then type `cmd` In the box. In Windows 8 and 10, you right-click on the windows icon, choose `Run` or equivalent in your local language, and type `cmd`. Alternatively you can type `Windows-key + R`. Now you should open a console window with a prompt showing something like Listing A.1.



(a) Linux



(b) MacOS X



(c) Windows

Figure A.1: The top file hierarchy levels of common operating systems.

Windows	MacOS X/Linux	Description
dir	ls	Show content of present directory.
cd <dir>	cd <dir>	Change present directory to <dir>.
mkdir <dir>	mkdir <dir>	create directory <dir>.
rmdir <dir>	rmdir <dir>	delete <dir> (Warning: cannot be reverted).
move <file> <file or dir>	mv <file> <file or dir>	Move <fil> to <file or dir>.
copy <file> <file>	cp <file> <file>	Create a new file called <file> as a copy of <file>.
del <file>	rm <file>	delete <file> (Warning: cannot be reverted).
echo <string or variable>	echo <string or variable>	Write a string or content of a variable to screen.

Table A.1: The most important console commands for Windows, MacOS X, and Linux.

Listing A.1: The Windows console.

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\sporring>
```

To see which files are in the directory, use ***dir*** as shown in Listing A.2. · dir

Listing A.2: Directory listing with dir.

```
C:\Users\sporring>dir
 Volume in drive C has no label.
 Volume Serial Number is 94F0-31BD

 Directory of C:\Users\sporring

30-07-2015  15:23    <DIR>          .
30-07-2015  15:23    <DIR>          ..
30-07-2015  14:27    <DIR>          Contacts
30-07-2015  14:27    <DIR>          Desktop
30-07-2015  17:40    <DIR>          Documents
30-07-2015  15:11    <DIR>          Downloads
30-07-2015  14:28    <DIR>          Favorites
30-07-2015  14:27    <DIR>          Links
30-07-2015  14:27    <DIR>          Music
30-07-2015  14:27    <DIR>          Pictures
30-07-2015  14:27    <DIR>          Saved Games
30-07-2015  17:27    <DIR>          Searches
30-07-2015  14:27    <DIR>          Videos
               0 File(s)              0 bytes
              13 Dir(s)  95.004.622.848 bytes free

C:\Users\sporring>
```

We see that there are no files and thirteen directories (DIR). The columns tell from left to right, the date and time of their creation, the file size or if it is a folder, and the name file or directory name. The first two folders "`.`" and "`..`" are found in each folder and refers to this folder as well as the one above in the hierarchy. In this case, the folder "`.`" is an alias for `C:\Users\tracking` and "`..`" for `C:\Users`.

Use ***cd*** to change directory, e.g., to `Documents` as in Listing A.3. · cd

Listing A.3: Change directory with cd.

```
C:\Users\sporring>cd Documents

C:\Users\sporring\Documents>
```

The directory `Documents` is that Windows Explorer selects, when you press the `Documents` short-cut in Explorer. 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. · mkdir

Listing A.4: Creating a directory with `mkdir`.

```
C:\Users\sporring\Documents>mkdir myFolder

C:\Users\sporring\Documents>dir
 Volume in drive C has no label.
 Volume Serial Number is 94F0-31BD

 Directory of C:\Users\sporring\Documents

30-07-2015  19:17    <DIR>          .
30-07-2015  19:17    <DIR>          ..
30-07-2015  19:17    <DIR>          myFolder
               0 File(s)              0 bytes
               3 Dir(s)  94.656.638.976 bytes free

C:\Users\sporring\Documents>
```

By using `dir` we inspect the result.

Files can be created by, e.g., ***`echo`*** and *redirection* as demonstrated in Listing A.5.

· `echo`
· redirection

Listing A.5: Creating a file with `echo` and redirection.

```
C:\Users\sporring\Documents>echo "Hi" > hi.txt

C:\Users\sporring\Documents>dir
 Volume in drive C has no label.
 Volume Serial Number is 94F0-31BD

 Directory of C:\Users\sporring\Documents

30-07-2015  19:18    <DIR>          .
30-07-2015  19:18    <DIR>          ..
30-07-2015  19:17    <DIR>          myFolder
30-07-2015  19:18                 8 hi.txt
               1 File(s)              8 bytes
               3 Dir(s)  94.656.634.880 bytes free

C:\Users\sporring\Documents>
```

To move the file `hi.txt` to the directory `myFolder` use ***`move`*** as shown in Listing A.6.

· `move`

Listing A.6: Move a file with `move`.

```
C:\Users\sporring\Documents>move hi.txt myFolder
        1 file(s) moved.

C:\Users\sporring\Documents>
```

Finally, use ***`del`*** to delete a file and ***`rmdir`*** to delete a directory as shown in Listing A.7.

· `del`
· `rmdir`

Listing A.7: Delete files and directories with `del` and `rmdir`.

```
C:\Users\sporring\Documents>cd myFolder

C:\Users\sporring\Documents\myFolder>del hi.txt

C:\Users\sporring\Documents\myFolder>cd ..

C:\Users\sporring\Documents>rmdir myFolder

C:\Users\sporring\Documents>dir
 Volume in drive C has no label.
 Volume Serial Number is 94F0-31BD

 Directory of C:\Users\sporring\Documents

30-07-2015  19:20    <DIR>          .
30-07-2015  19:20    <DIR>          ..
               0 File(s)              0 bytes
               2 Dir(s)  94.651.142.144 bytes free

C:\Users\sporring\Documents>
```

The commands available from the console must be in its *search path*. The search path can be seen using `echo` as shown in Listing A.8. · search path

Listing A.8: Displaying the search path.

```
C:\Users\sporring\Documents>echo %Path%
C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;
   C:\Windows\System32\WindowsPowerShell\v1.0\;"\Program
   Files\emacs-24.5\bin\"

C:\Users\sporring\Documents>
```

The path can be changed using the Control panel in the graphical user interface. In Windows 7, choose the Control panel, choose `System and Security` → `System` → `Advanced system settings` → `Environment Variables`. In Windows 10 you can find this window by searching for "Environment" in the Control panel. In the window's`System variables` box, double-click on `Path` and add or remove a path from the list. The search path is a list of paths separated by ";". Beware, Windows uses the search path for many different tasks, so remove only paths that you are certain are not used for anything.

A useful feature of the console is that you can use the `tab`-key to cycle through filenames. E.g., if you write `cd` followed by a space and `tab` a couple of times, then the console will suggest 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 are shown in Table A.1. In MacOS X, you find the console by opening · `bash`

`Finder` and navigating to `Applications` → `Utilities` -> `Terminal`. In Linux, the console can be started by typing `Ctrl + Alt + T`. Some Linux distributions have other key-combinations such as `Super + T`.

Once opened, the console is shown in a window with content as shown in Listing A.9.

Listing A.9: The Windows console.

```
Last login: Thu Jul 30 11:52:07 on ttys000
FN11194:~ sporring$
```

"FN11194" is the name of the computer, the character ∼ is used as an alias for the user's home directory, and "sporring" is the username for the user presently logged onto the system. Use *`ls`* to see which files are present, as shown in Listing A.10. · `ls`

Listing A.10: Display a directory content with `ls`.

```
FN11194:~ sporring$ ls
Applications    Documents       Library         Music           Public
Desktop         Downloads       Movies          Pictures
FN11194:~ sporring$
```

More details about the files are available by using flags to `ls` as demonstrated in Listing A.11.

Listing A.11: Display extra information about files using flags to `ls`.

```
FN11194:~ sporring$ ls -l
drwx------   6 sporring  staff   204 Jul 30 14:07 Applications
drwx------+ 32 sporring  staff  1088 Jul 30 14:34 Desktop
drwx------+ 76 sporring  staff  2584 Jul  2 15:53 Documents
drwx------+  4 sporring  staff   136 Jul 30 14:35 Downloads
drwx------@ 63 sporring  staff  2142 Jul 30 14:07 Library
drwx------+  3 sporring  staff   102 Jun 29 21:48 Movies
drwx------+  4 sporring  staff   136 Jul  4 17:40 Music
drwx------+  3 sporring  staff   102 Jun 29 21:48 Pictures
drwxr-xr-x+  5 sporring  staff   170 Jun 29 21:48 Public
FN11194:~ sporring$
```

The flag `-l` means long, and many other flags can be found by querying the built-in manual with `man ls`. The output is divided into columns, where the left column shows a number of codes: "d" means they are directory followed by three sets of optional "rwx" denoting whether the owner, the associated group of users and anyone can "r" - read, "w" - write, or "x" - execute the files. In this case, 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 `Documents` as shown in Listing A.12. · `cd`

Listing A.12: Change directory with `cd`.

```
FN11194:~ sporring$ cd Documents/
FN11194:Documents sporring$
```

Note that some graphical user interfaces translate standard filenames and directories to the local language, such that navigating using the graphical user interface will reveal other files and directories, which, however, are aliases.

You can yourself create a new directory using ***mkdir*** as demonstrated in Listing A.13. · `mkdir`

Listing A.13: Creating a directory using `mkdir`.

```
FN11194:Documents sporring$  mkdir myFolder
FN11194:Documents sporring$ ls
myFolder
FN11194:tmp sporring$
```

A file can be created using ***echo*** and with *redirection* as shown in Listing A.14. · `echo` · redirection

Listing A.14: Creating a file with `echo` and redirection.

```
FN11194:Documents sporring$ echo "hi" > hi.txt
FN11194:Documents sporring$ ls
hi.txt          myFolder
```

To move the file `hi.txt` into `myFolder`, use ***mv***. This is demonstrated in Listing A.15. · `mv`

Listing A.15: Moving files with `mv`.

```
FN11194:Documents sporring$ echo mv hi.txt myFolder/
FN11194:Documents sporring$
```

To delete the file and the directory, use ***rm*** and ***rmdir*** as shown in Listing A.16. · `rm` · `rmdir`

Listing A.16: The Windows console.

```
FN11194:Documents sporring$ cd myFolder/
FN11194:myFolder sporring$ rm hi.txt
FN11194:myFolder sporring$ cd ..
FN11194:Documents sporring$ rmdir myFolder/
FN11194:Documents sporring$ ls
FN11194:Documents sporring$
```

Only commands found on the *search-path* are available in the console. The content of the search-path is seen using the `echo` command as demonstrated in Listing A.17. · search-path

Listing A.17: The content of the search-path.

```
FN11194:Documents sporring$ echo $PATH
/Applications/Maple 17/:/Applications/PackageMaker.app/Contents/MacOS/:
   /Applications/MATLAB_R2014b.app/bin/:/opt/local/bin:/opt/local/sbin:
   /usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/opt/X11/bin:
   /Library/TeX/texbin
FN11194:Documents sporring$
```

The search-path can be changed by editing the setup file for Bash. On MacOS X it is called `~/.profile`, and on Linux it is either `~/.bash_profile` or `~/.bashrc`. Here new paths can be added by adding the following line: `export PATH="<new path>:<another new path>:$PATH"`.

A useful feature of Bash is that the console can help you write commands. E.g., if you write `fs` followed by pressing the `tab`-key, and if Mono is in the search-path, then Bash will typically respond by completing the line as `fsharp`, and by further pressing the `tab`-key some times, Bash will show the list of options, typically `fshpari` and `fsharpc`. Also, most commands have an extensive manual, which can be accessed using the `man` command. E.g., the manual for `rm` is retrieved by `man rm`.

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 means that for a number consisting 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:

· decimal number
· base
· decimal point
· digit

$$v = \sum_{i=-m}^{n} d_i 10^i \tag{B.1}$$

The basic unit of information in almost all computers is the binary digit or *bit* for short. A *binary number* consists of a sequence of binary digits separated by a decimal point, where each digit can have values $b \in \{0, 1\}$, and the base is 2. The general equation is,

· bit
· binary number

$$v = \sum_{i=-m}^{n} b_i 2^i \tag{B.2}$$

and examples are $1011.1_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} = 11.5$. Notice that we use subscript 2 to denote a binary number, while no subscript is used for decimal numbers. The left-most bit is called the *most significant bit*, and the right-most bit is called the *least significant bit*. Due to typical organisation of computer memory, 8 binary digits is called a *byte*, and 32 digits a *word*.

· most significant bit
· least significant bit
· byte
· word

Other number systems are often used, e.g., *octal numbers*, which are base 8 numbers, where each digit is $o \in \{0, 1, \ldots, 7\}$. Octals are useful short-hand for binary, since 3 binary digits maps 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 octal digits. Thus $367 = 101101111_2 = 557_8 = 16f_{16}$. A list of the integers 0–63 is various bases is given in Table B.1.

· octal number
· hexadecimal number

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 between any 2 given numbers there are infinitely many numbers. Reals are widely used for calculation, but since any computer only has finite memory, it is impossible to represent all possible reals on one. 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

· reals
· IEEE 754 double precision floating-point format
· binary64
· double

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.

using 64 bits. These bits are divided into 3 parts: sign, exponent and fraction,

$$s\, e_1 e_2 \dots e_{11}\, m_1 m_2 \dots m_{52},$$

where s, e_i, and m_j are binary digits. The bits are converted to a number using the equation by first calculating the exponent e and the mantissa m,

$$e = \sum_{i=1}^{11} e_i 2^{11-i}, \tag{B.3}$$

$$m = \sum_{j=1}^{52} m_j 2^{-j}. \tag{B.4}$$

I.e., the exponent is an integer, where $0 \leq e < 2^{11}$, and the mantissa is a rational, where $0 \leq m < 1$. For most combinations of e and m the real number v is calculated as,

$$v = (-1)^s\, (1 + m)\, 2^{e-1023} \tag{B.5}$$

with the exception that

	$m = 0$	$m \neq 0$
$e = 0$	$v = (-1)^s\, 0$ (signed zero)	$v = (-1)^s\, m2^{1-1023}$ (subnormals)
$e = 2^{11} - 1$	$v = (-1)^s \infty$	$v = (-1)^s$ NaN (not a number)

· subnormals
· NaN
· not a number

where $e = 2^{11} - 1 = 11111111111_2 = 2047$. The largest and smallest number that is not infinity is thus

$$e = 2^{11} - 2 = 2046 \tag{B.6}$$

$$m = \sum_{j=1}^{52} 2^{-j} = 1 - 2^{-52} \simeq 1. \tag{B.7}$$

$$v_{\max} = \pm\left(2 - 2^{-52}\right) 2^{1023} \simeq \pm 2^{1024} \simeq \pm 10^{308} \tag{B.8}$$

The density of numbers varies in such a way that when $e - 1023 = 52$, then

$$v = (-1)^s \left(1 + \sum_{j=1}^{52} m_j 2^{-j}\right) 2^{52} \tag{B.9}$$

$$= \pm\left(2^{52} + \sum_{j=1}^{52} m_j 2^{-j} 2^{52}\right) \tag{B.10}$$

$$= \pm\left(2^{52} + \sum_{j=1}^{52} m_j 2^{52-j}\right) \tag{B.11}$$

$$\overset{k=52-j}{=} \pm\left(2^{52} + \sum_{k=51}^{0} m_{52-k} 2^{k}\right) \tag{B.12}$$

which are all integers in the range $2^{52} \leq |v| < 2^{53}$. When $e - 1023 = 53$, then the same calculation gives

$$v \overset{k=53-j}{=} \pm\left(2^{53} + \sum_{k=52}^{1} m_{53-k} 2^{k}\right) \tag{B.13}$$

which are every second integer in the range $2^{53} \leq |v| < 2^{54}$, and so on for larger values of e. When $e - 1023 = 51$, the same calculation gives,

$$v \overset{k=51-j}{=} \pm\left(2^{51} + \sum_{k=50}^{-1} m_{51-k} 2^{k}\right) \tag{B.14}$$

which is a distance between numbers of $1/2$ in the range $2^{51} \leq |v| < 2^{52}$, and so on for smaller values of e. Thus we may conclude that the distance between numbers in the interval $2^n \leq |v| < 2^{n+1}$ is 2^{n-52}, for $-1022 = 1 - 1023 \leq n < 2046 - 1023 = 1023$. For subnormals, the distance between numbers are

$$v = (-1)^s \left(\sum_{j=1}^{52} m_j 2^{-j}\right) 2^{-1022} \tag{B.15}$$

$$= \pm\left(\sum_{j=1}^{52} m_j 2^{-j} 2^{-1022}\right) \tag{B.16}$$

$$= \pm\left(\sum_{j=1}^{52} m_j 2^{-j-1022}\right) \tag{B.17}$$

$$\overset{k=-j-1022}{=} \pm\left(\sum_{j=-1023}^{-1074} m_{-k-1022} 2^{k}\right) \tag{B.18}$$

which gives a distance between numbers of $2^{-1074} \simeq 10^{-323}$ in the range $0 < |v| < 2^{-1022} \simeq 10^{-308}$.

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', which is shared by many other European languages, but which have other 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 contains 106,230 different characters albeit only 2,600 are included in the official Chinese language test at highest level.

Presently, the most common character set used is Unicode Transformation Format (UTF), whose most popular encoding schemes are 8-bit (UTF-8) and 16-bit (UTF-16). Many other character sets exist, and many of the later build on the American Standard Code for Information Interchange (ASCII). The ISO-8859 codes were an intermediate set of character sets that are still in use, but which is greatly inferior to UTF. Here we will briefly give an overview of ASCII, ISO-8859-1 (Latin1), and UTF.

C.1 ASCII

The *American Standard Code for Information Interchange* (*ASCII*) [8], is a 7 bit code tuned for the letters of the english language, numbers, punctuation symbols, control codes and space, see Tables C.1 and C.2. The first 32 codes are reserved for 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.

· American Standard Code for Information Interchange
· ASCII

The code order is known as *ASCIIbetical order*, and it is sometimes used to perform arithmetic on codes, e.g., an upper case letter with code c may be converted to lower case by adding 32 to its code. The ASCIIbetical order also has consequence for sorting, i.e., when sorting characters according to their ASCII code, 'A' comes before 'a', which comes before the symbol '{'.

· ASCIIbetical order

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 graphic 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 are the same in ISO 8859-1. Table C.3 shows the characters above 7e. Codes 00-1f and 7f-9f are undefined in ISO 8859-1.

· Latin1

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

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. However, not all abstract characters require a unit of several code points to be specified. Code points are divided into 17 planes each with $2^{16} = 65,536$ code points. Planes are further subdivided into named *blocks*. The first plane is called the *Basic Multilingual plane* and its block of the first 128 code points is called the *Basic Latin block* and are identical to ASCII, see Table C.1, and code points 128-255 is called the *Latin-1 Supplement block*, and are identical to the upper range of ISO 8859-1, see Table C.3. Each code-point has a number of attributes such as the *Unicode general category*. Presently more than 128,000 code points are defined covering 135 modern and historic writing systems, and obtained at `http://www.unicode.org/Public/UNIDATA/UnicodeData.txt`, which includes the code point, name, and general category.

· Unicode Standard
· code point
· blocks
· Basic Multilingual plane
· Basic Latin block
· Latin-1 Supplement block
· Unicode 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 it is visualized as ’A’. More digits are used for code points of the remaining planes.

The general category is used to specify valid characters that not necessarily have a visual representation but possibly transforms text. Some categories and their letters in the first 256 code points are shown in Table C.5.

To store and retrieve code points, they must be encoded and decoded. A common encoding is *UTF-8*, which encodes code points as 1 to 4 bytes, and which is backward-compatible with ASCII and ISO 8859-1. Hence, in all 3 coding systems the character with code 65 represents the character ’A’. Another popular encoding scheme is *UTF-16*, which encodes characters as 2 or 4 bytes, but which is not backward-compatible with ASCII or ISO 8859-1. UTF-16 is used internally in many compilers, interpreters and operating systems.

· UTF-8
· UTF-16

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.

x0+0x	80	90	A0	B0	C0	D0	E0	F0
00			NBSP	°	À	Ð	à	ð
01			¡	±	Á	Ñ	á	ñ
02			¢	^{2}	Â	Ò	â	ò
03			£	^{3}	Ã	Ó	ã	ó
04			¤	´	Ä	Ô	ä	ô
05			¥	μ	Å	Õ	å	õ
06			¦	¶	Æ	Ö	æ	ö
07			§	·	Ç	×	ç	÷
08			¨	¸	È	Ø	è	ø
09			©	^{1}	É	Ù	é	ù
0a			ª	º	Ê	Ú	ê	ú
0b			«	»	Ë	Û	ë	û
0c			¬	$\frac{1}{4}$	Ì	Ü	ì	ü
0d			SHY	$\frac{1}{2}$	Í	Ý	í	ý
0e			®	$\frac{3}{4}$	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.

General category	Code points	Name
Lu	U+0041–U+005A, U+00C0–U+00D6, U+00D8–U+00DE	Upper case letters
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.

D | Common Language Infrastructure

The *Common Language Infrastructure* (*CLI*), not to be confused with *Command Line Interface* with the same acronym, is a technical standard developed by Microsoft [4, 3]. The standard specifies a language, its format, and a runtime environment that can execute the code. The main feature is that it provides a common interface between many languages and many platforms, such that programs can collaborate in a language agnostic manner and can be executed on different platforms without having to be recompiled. Main features of the standard are:

· Common Language Infrastructure
· CLI
· Command Line Interface

Common Type System (CTS) which defines a common set of types that can be used across different languages as if it were their own.

· Common Type System
· CTS

Metadata which defines a common method for referencing programming structures such as values and functions in a language independent manner.

· Metadata

Common Intermediate Language (CIL) which is a platform-independent, stack-based, object-oriented assembly language that can be executed by the Virtual Execution System.

· Common Intermediate Language
· CIL

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.

· Virtual Execution System
· VES
· Common Language Runtime
· CLR
· just-in-time

The process of running an F# program is shown in Figure D.1[1]: First the F# code is compiled or interpreted to CIL. This code possibly combined with other CIL code is then converted to a machine-readable code, and the result is then executed on the platform.

CLI defines a *module* as a single file containing executable code by VES. Hence, CLI's notion of 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.

· module
· manifest
· assembly

[1] Jon: **update figure**

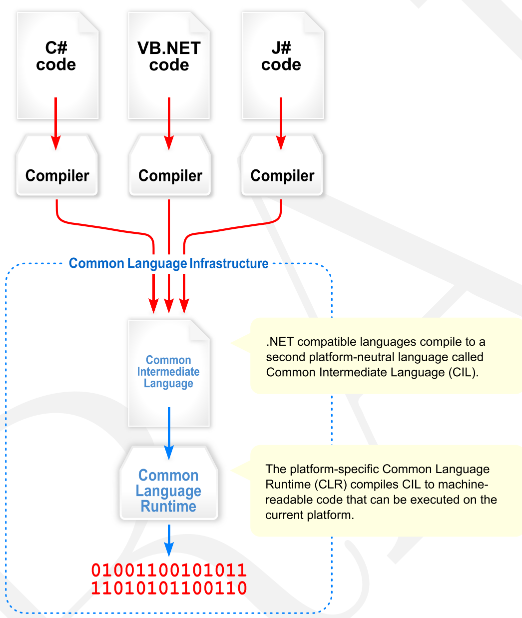


Figure D.1: Visual overview of CLI. Figure by Jarkko Piiroinen - Own work, Public Domain, `https://commons.wikimedia.org/w/index.php?curid=3602584`.

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 2.0. `http://www.omg.org/spec/UML/2.0/`.

[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