Learning to Program with F#

Jon Sporring

Department of Computer Science,
University of Copenhagen

September 7, 2018

Contents

1. Preface **5**

2. Introduction **6**
2.1. How to Learn to Solve Problems by Programming 6
2.2. How to Solve Problems 7
2.3. Approaches to Programming 8
2.4. Why Use F# 9
2.5. How to Read This Book 9

3. Executing F# code **11**
3.1. Source code 11
3.2. Executing programs 12

4. Quick-start guide **15**

5. Using F# as a calculator **21**
5.1. Literals and basic types 21
5.2. Operators on basic types 27
5.3. Boolean arithmetic 29
5.4. Integer arithmetic 30
5.5. Floating point arithmetic 33
5.6. Char and string arithmetic 34
5.7. Programming intermezzo: Hand conversion between decimal and binary numbers 36

6. Values and functions **37**
6.1. Value bindings 40
6.2. Function bindings 45
6.3. Operators 52
6.4. Do bindings 54
6.5. The Printf function 54
6.6. Reading from the console 57
6.7. Variables 58
6.8. Reference cells 61
6.9. Tuples 64

7. In-code documentation **95**

8. Controlling program flow **103**
8.1. While and for loops 103
8.2. Conditional expressions 110
8.3. Programming intermezzo: Automatic conversion of decimal to binary numbers 113

9. Organising code in libraries and application programs **116**
9.1. Modules . . . 116
9.2. Namespaces . . . 121
9.3. Compiled libraries . . . 124

10. Testing programs **129**
10.1. White-box testing . . . 133
10.2. Black-box testing . . . 138
10.3. Debugging by tracing . . . 142

11. Collections of data **153**
11.1. Strings . . . 153
11.1.1. String properties . . . 154
11.1.2. String module . . . 154
11.2. Lists . . . 157
11.2.1. List properties . . . 162
11.2.2. List module . . . 162
11.3. Arrays . . . 168
11.3.1. Array properties and methods . . . 171
11.3.2. Array module . . . 172
11.4. Multidimensional arrays . . . 179
11.4.1. Array2D module . . . 183

12. The imperative programming paradigm **187**
12.1. Imperative design . . . 188

13. Recursion **190**
13.1. Recursive functions . . . 190
13.2. The call stack and tail recursion . . . 193
13.3. Mutual recursive functions . . . 197

14. Programming with types **203**
14.1. Type abbreviations . . . 203
14.2. Enumerations . . . 204
14.3. Discriminated Unions . . . 205
14.4. Records . . . 209
14.5. Structures . . . 213
14.6. Variable types . . . 215

15. Pattern matching **219**
15.1. Wildcard pattern . . . 223
15.2. Constant and literal patterns . . . 224
15.3. Variable patterns . . . 226
15.4. Guards . . . 227
15.5. List patterns . . . 228
15.6. Array, record, and discriminated union patterns . . . 229
15.7. Disjunctive and conjunctive patterns . . . 232
15.8. Active Pattern . . . 234
15.9. Static and dynamic type pattern . . . 238

16. Higher order functions **241**
16.1. Function composition . . . 244
16.2. Currying . . . 245

17. The functional programming paradigm **247**
17.1. Functional design 249

18. Handling Errors and Exceptions **251**
18.1. Exceptions 251
18.2. Option types 265
18.3. Programming intermezzo: Sequential division of floats 267

19. Working with files **271**
19.1. Command line arguments 272
19.2. Interacting with the console 274
19.3. Storing and retrieving data from a file 277
19.4. Working with files and directories. 284
19.5. Reading from the internet 284
19.6. Resource Management 287
19.7. Programming intermezzo: Ask user for existing file 289

20. Classes and objects **291**
20.1. Constructors and members 292
20.2. Accessors 295
20.3. Objects are reference types 299
20.4. Static classes 301
20.5. Recursive members and classes 303
20.6. Function and operator overloading 304
20.7. Additional constructors 307
20.8. Interfacing with `printf` family 310
20.9. Programming intermezzo 311

21. Derived classes **317**
21.1. Inheritance 317
21.2. Abstract class 322
21.3. Interfaces 325
21.4. Programming intermezzo: Chess 327

22. The object-oriented programming paradigm **343**
22.1. Identification of objects, behaviors, and interactions by nouns-and-verbs . . 345
22.2. Class diagrams in the Unified Modelling Language 345
22.3. Programming intermezzo: designing a racing game 350

23. Graphical User Interfaces **356**
23.1. Opening a window 357
23.2. Drawing geometric primitives 359
23.3. Programming intermezzo: Hilbert Curve 371
23.4. Handling events 378
23.5. Labels, buttons, and pop-up windows 382
23.6. Organising controls 387

24. The Event-driven programming paradigm **396**

25. Where to go from here **397**

A. The Console in Windows, MacOS X, and Linux **400**
A.1. The Basics 400

A.2. Windows 400
A.3. MacOS X and Linux 404

B. Number Systems on the Computer **408**
B.1. Binary Numbers 408
B.2. IEEE 754 Floating Point Standard 408

C. Commonly Used Character Sets **412**
C.1. ASCII 412
C.2. ISO/IEC 8859 413
C.3. Unicode 413

D. Common Language Infrastructure **424**

E. Language Details **426**
E.1. Arithmetic operators on basic types 426
E.2. Basic arithmetic functions 429
E.3. Precedence and associativity 431

Bibliography **433**

Index **434**

1 | Preface

This book has been written as an introduction to programming for novice programmers. It is used in the first programming course at the University of Copenhagen's bachelor in computer science program. It has been typeset in LaTeX, and all programs have been developed and tested in Mono version 5.10.1.57.

This book started as a few chapters in 2016 and was to a large extent completed in 2017. This book was developed alongside the course Programmering og Problemløsning (programming and problem solving) and I am very thankful for the positive feedback and suggestions numerous people have given me. I would particularly like to thank Malthe Sporring for his insightful and detailed comments to every (!) page of this book. I also would like to acknowledge the invaluable feedback from my co-teachers: Torben Mogensen, Martin Elsmann, Christina Lioma; my teaching assistants: Sune Hellfritzsch, Emil Bak, Jesper Erno, Rasmus Johannesson, Jan Rolandsen, Peter Pedersen, Joachim Tilsted Kristensen, Lukas Svarre Engedal, Matthias Brix, Kristian Fogh Nissen, Emil Petersen, Jens Larsen, Emil Bak, Lasse Grønborg, Mads Obitsø, Maurits Pallesen, Tor Skovsgaard, Baldar Ivarsen, Alexander Christensen, Lars-Bo Nielsen, Frederik Schmidt, Lukas Engedal, Jan Rolandsen. And finally, thanks to all the students of our course who have had the patience and endurance to labor and enjoy learning to program using F#.

Jon Sporring
Associate Professor, Ph.d.
Department of Computer Science,
University of Copenhagen
September 7, 2018

2 | Introduction

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

2.1. How to Learn to Solve Problems by Programming

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

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

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

3. Practice: In order to be a good programmer, the most essential step is: practice, practice, practice! It has been estimated that to master anything, then you have to have spent at least 10000 hours practicing, so get started logging hours! It of course matters, how you practice. This book teaches a number of different programming themes. The point is that programming is thinking, and the scaffold you use shapes

your thoughts. It is therefore important to recognize this scaffold and to have the ability to choose one which suits your ideas and your goals best. The best way to expand your abilities is to sharpen your present abilities, push yourself into new territory, and try something new. Do not be afraid to make errors or be frustrated at first. These are the experiences that make you grow.

4. Solve real problems: I have found that using my programming skills in real situations with customers demanding specific solutions, has forced me to put the programming tools and techniques that I use into perspective. Sometimes a task requires a cheap and fast solution, other times customers want a long-perspective solution with bug fixes, upgrades, and new features. Practicing solving real problems helps you strike a balance between the two when programming. It also allows makes you a more practical programmer, by allowing you to recognize its applications in your everyday experiences. Regardless, real problems create real programmers.

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, given by George Pólya [9] and adapted to programming, is:

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

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

Implement the plan: Implementation is the act of transforming a program design into code. A crucial part of any implementation is choosing which programming language to use. Furthermore, the solution to many problems will have a number of implementations which vary in how much code they require, to which degree they rely on external libraries, which programming style they are best suited for, what machine resources they require, and how long time they take to run on a computer. With a good design, the coding is usually easy, since the design will have uncovered the major issues and found solutions for these, but sometimes implementation reveals new problems, which require rethinking the design. Most often the implementation step also require a careful documentation of key aspects of the code, e.g., a user manual for the user, and internal notes for fellow programmers that are to maintain and update the code in the future.

Reflect on the result: A crucial part of any programming task is ensuring that the program solves the problem sufficiently. Ask yourself questions such as: What are the program's errors, is the documentation of the code sufficient and relevant for its intended use? Is the code easily maintainable and extendable by other programmers? Which parts of your method would you avoid or replicate in future programming sessions? Can you reuse some of the code you developed in other programs?

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

2.3. Approaches to Programming

This book focuses on several fundamentally different approaches to programming:

· imperative programming

Imperative programming emphasizes *how a program shall accomplish a solution* and focusses less on *what the solution is*. A cooking recipe is an example of the spirit of imperative programming, where the recipe emphasizes what should be done in each step rather than describing the result. For example, a recipe for bread might tell you to first mix yeast and water, then add flour, etc. In imperative programming what should be done are called *statements* and in the recipe analogy, the steps are the statements. Statements influence the computer's *states*, in the same way that adding flour changes the state of our dough. Almost all computer hardware is designed to execute low-level programs written in imperative style. Imperative programming builds on the Turing machine [10]. As a historical note, the first major language was FORTRAN [6] which emphasized an imperative style of programming.

· statement

· state

· declarative programming

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

· functional programming

· function

· expression

· structured programming

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

· Object-oriented programming

· object

· event-driven programming

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

· 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 its advantages and disadvantages.

2.4. Why Use F#

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

Interactive and compile mode: F# has an interactive and a compile mode of operation. In interactive mode you can write code that is executed immediately in a manner similar to working with a calculator, while in compile mode you combine many lines of code possibly in many files into a single application, which is easier to distribute to people who are not F# experts and is faster to execute.

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

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

Multi-platform: F# is available on Linux, Mac OS X, Android, iOS, Windows, GPUs, and browsers 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) organized as assemblies. Thus, if you find that certain parts of a program are easy to express in F# and others in C++, then you will be able to combine these parts later into a single program.

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

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

2.5. How to Read This Book

Learning to program requires mastering a programming language, however, most programming languages contain details that are rarely used or used in contexts far from a specific programming topic. Hence, this book only includes a subset of F# but focuses on language structures necessary to understand several common programming paradigms: Imperative programming mainly covered in Chapters 6 to 11, functional programming mainly covered

in Chapters 13 to 16, object-oriented programming in Chapters 20 and 22, and event-driven programming in Chapter 23. A number of general topics are given in the appendix for reference. 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 come in many forms, which are identified by suffixes. The *source code* is an F# program written in human-readable form using an editor. F# recognizes the following types of source code files:

· 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

· scripts
· script-fragment

programs, which will be discussed in detail in Chapter 9.

3.2. Executing programs

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

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

Listing 3.1 gettingStartedStump.fsx: A simple demonstration script.

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

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

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

Listing 3.2: An interactive session.

```
1  $ fsharpi
2
3  F# Interactive for F# 4.1 (Open Source Edition)
4  Freely distributed under the Apache 2.0 Open Source License
5
6  For help type #help;;
7
8  > let a = 3.0
9  - do printfn "%g" a;;
10 3
11
12 val a : float = 3.0
13 val it : unit = ()
14
15 > #quit;;
```

We see that after typing `fsharpi`, then the program starts by stating details about itself

[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 the end of a script-fragment. Such possibly compound characters are called lexemes."**

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 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`, 1.88s $\gg$ 0.05s.

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

· 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 the 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

- expression
- `let`
- keyword
- binding
- let-binding
- integer number
- do-binding
- `do`
- statements

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

· `printfn`
· function
· `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 need 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 different to integer numbers

· decimal point
· floating point 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.

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

```
let a = 357.6
let b = 863.4
let c = a + b
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 an 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 are 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

· error message

· debugging

the application.

F# is a functional first programming language, and one implication of this is that names have a *lexical scope*. A scope is the lines in a program, where a binding is valid, and lexical scope means that to find the value of a name F# looks for the value in the above lines. 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

· encapsulate

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

Type inference in F# may cause problems since the type of a function is inferred 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.

```
1  val sum : x:int -> y:int -> int
2
3  > let c = sum 357.6 863.4;;
4    let c = sum 357.6 863.4;;
5    ------------^^^^^
6
7  stdin(3,13): error FS0001: This expression was expected to
     have type
8      'int'
9  but here has type
10     '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.

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

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

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 paren-

theses 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.

· `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 a description of integer see Appendix B.1, for floating point numbers see Appendix B.2, for ASCII and Unicode characters see Appendix C, for objects see Chapter 20, and for exceptions see Chapter 18.

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

Humans like to use the *decimal number* system for representing numbers. Decimal numbers are *base* 10, which means that a value is represented as two sequences of decimal digits separated by a *decimal point*, where each *digit* d has a position and a value $d \in \{0, 1, 2, \ldots, 9\}$. The part before the decimal point is called the *whole part* and the part after is called the *fractional part* of the number. 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 are all represented as bits, hence F# has a strong support for binary numbers. A *binary number* consists of a sequence of binary digits separated by a decimal point, where each digit can have values $b \in \{0, 1\}$, and the base is 2. E.g., the binary number $101.01_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 5.25$. Subscripts are often used to indicate the base of a number, e.g., 101.01_2 and 101.01_{10} are different numbers. Since base 10 is so common, the subscript for base 10 numbers is often omitted.

· bit
· binary 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.

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 use 16 as basis, and where each hexadecimal digit can be written in binary using exactly 4 bits. The hexadecimal digits use `0`–`9` to represent the values 0–9 and `a`–`f` in lower or alternatively upper case to represent the values 10-15. Thus, Octals and hexadecimals conveniently serve as shorthand for the much longer binary representation. As examples, the octal number 37_8 is $3 \cdot 8^1 + 7 \cdot 8^0 = 31$, and the hexadecimal number $f3_{16}$ is $15 \cdot 16^1 + 3 \cdot 16^0 = 243$.

· octal number
· hexadecimal number

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

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 or 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

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 5.4: 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
```

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 exist 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 cast 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 are 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 the following:

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 take only one argument and have the syntax:

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) +`

· precedence

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

`0.5`, which is erroneous since the addition of an integer with a float is undefined. This is an example of precedence, i.e., function evaluation takes precedence over addition meaning that it is performed before addition. Consider the arithmetic expression,

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

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

Table 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 have the same precedence. Full table is given in Table E.5.

Associativity implies the order in which calculations are performed for operators of the 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.

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.

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 rules by making a simple script.**

Advice

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 the 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 result translates back to the Boolean value false. In F# the truth table

- and
- or
- not
- truth table

for the basic Boolean operators can be produced by a program as shown in Listing 5.13.

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. The difference between verbose 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 straightforward. 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 leftmost bit is interpreted differently for signed and unsigned integers, which gives rise to potential overflow errors.

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

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

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 the 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 than 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 lines of the error message are the most important parts, which tells us what exception was cast and why the program stopped. The middle is 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 as the built-in function `pown`. This function is demonstrated in Listing 5.20 for calculating 2^5.

Listing 5.20: Integer exponent function.

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

For binary arithmetic on integers, the following operators are available: `<leftExpr> <<< <rightExpr>`, which shifts the bit pattern of `<leftExpr>` `<rightExpr>` positions to the left

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

Table 5.6.: Boolean exclusive or truth table.

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 the 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 a number close to but not identical to what `357.9` is represented as, and thus, when subtracting these two representations, we get a very small 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 an integer. A typical example is the conversion of character 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

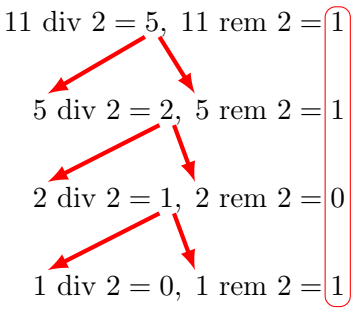
`"abs" = "abs"` is true. The "<>" operator is the boolean negation of the "=" operator, e.g., `"abs" <> "absalon"` is true, while `"abs" <> "abs"` is false. For the "<" , "<=", ">", and ">=" operators, the strings are ordered alphabetically, such that `"abs" < "absalon" && "absalon" < "milk"` is true, 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 straightforward 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:



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

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

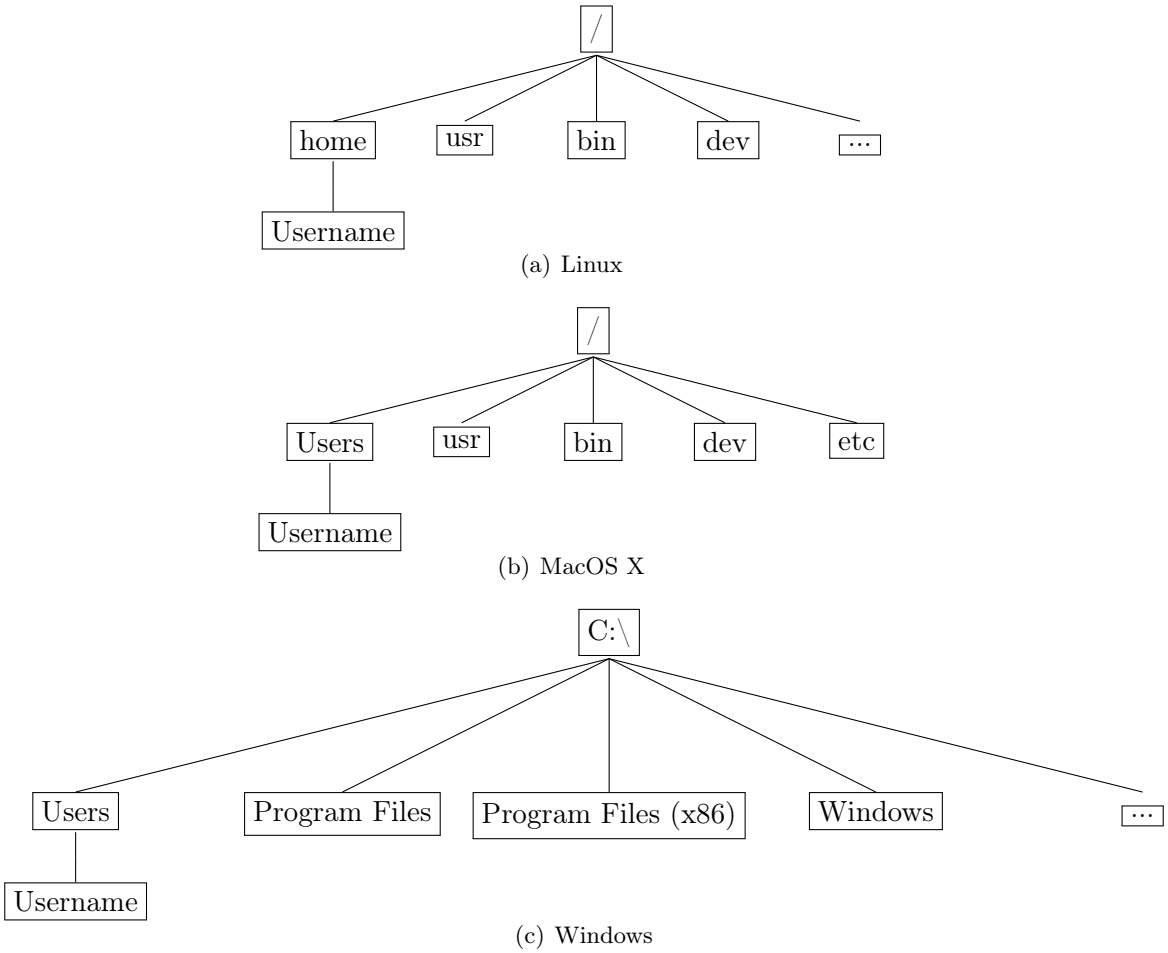
When you open a *directory* or *folder* in your preferred operating system, the directory will have a location in the file system, whether from the console or through the operating system's graphical user interface. The console will almost always be associated with a particular directory or folder in 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, MacOS 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 also make your own. In the following sections, we will review the most important commands in the three different operating systems. These are summarized in Table A.1.

A.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 <file1> <file2>`	`cp <file1> <file2>`	Create a new file called `<file2>` as a copy of `<file1>`.
`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 refer to this folder as well as the one above in the hierarchy. In this case, the folder "." is an alias for `C:\Users\sporring` and ".." for `C:\Users`.

Use *cd* to change directory, e.g., to `Documents`, as in Listing A.3. · cd

Listing A.3: Change directory with cd.

```
C:\Users\sporring>cd Documents

C:\Users\sporring\Documents>
```

Note that some systems translate default filenames, so their names may be given different names in different languages in the graphical user interface as compared to the console.

You can use *mkdir* to create a new directory called, e.g., `myFolder`, as illustrated in List- · mkdir

ing A.4.

Listing A.4: Creating a directory with `mkdir`.

```
C:\Users\sporring\Documents>mkdir myFolder

C:\Users\sporring\Documents>dir
 Volume in drive C has no label.
 Volume Serial Number is 94F0-31BD

 Directory of C:\Users\sporring\Documents

30-07-2015  19:17    <DIR>          .
30-07-2015  19:17    <DIR>          ..
30-07-2015  19:17    <DIR>          myFolder
               0 File(s)              0 bytes
               3 Dir(s)  94.656.638.976 bytes free

C:\Users\sporring\Documents>
```

By using `dir` we inspect the result.

Files can be created by, e.g., *`echo`* and *redirection*, as demonstrated in Listing A.5.

- `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 to you the available directories.

A.3. MacOS X and Linux

MacOS X (OSX) and Linux are very similar, and both have the option of using ***`bash`*** as console. It is in the standard console on MacOS X and on many Linux distributions. A summary of the most important `bash` commands is shown in Table A.1. In MacOS X,

· `bash`

you find the console by opening `Finder` and navigating to `Applications` → `Utilities` -> `Terminal`. In Linux, the console can be started by typing `Ctrl + Alt + T`. Some Linux distributions have other key-combinations such as `Super + T`.

Once opened, the console is shown in a window with content, as shown in Listing A.9.

Listing A.9: The MacOS console.

```
Last login: Thu Jul 30 11:52:07 on ttys000
FN11194:~ sporring$
```

"FN11194" is the name of the computer, the character ~ is used as an alias for the user's home directory, and "sporring" is the username for the user presently logged onto the system. Use *ls* to see which files are present, as shown in Listing A.10. ·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" stands for directory, and the set of three of optional "rwx" denote whether respectively the owner, the associated group of users, and anyone can respectively "r" - read, "w" - write, and "x" - execute the file. In all directories but the `Public` directory, only the owner can do any of the three. For directories, "x" means permission to enter. The second column can often be ignored, but shows how many links there are to the file or directory. Then follows the username of the owner, which in this case is `sporring`. The files are also associated with a group of users, and in this case, they all are associated with the group called `staff`. Then follows the file or directory size, the date of last change, and the file or directory name. There are always two hidden directories: "`.`" and "`..`", where "`.`" is an alias for the present directory, and "`..`" for the directory above. Hidden files will be shown with the `-a` flag.

Use *cd* to change to the directory, for example to `Documents` as shown in Listing A.12. · 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 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: Deleting files and directories.

```
FN11194:Documents sporring$ cd myFolder/
FN11194:myFolder sporring$ rm hi.txt
FN11194:myFolder sporring$ cd ..
FN11194:Documents sporring$ rmdir myFolder/
FN11194:Documents sporring$ ls
FN11194:Documents sporring$
```

Only commands found on the *search-path* are available in the console. The content of the search-path is seen using the `echo` command, as demonstrated in Listing A.17. · search-path

Listing A.17: The content of the search-path.

```
1 FN11194:Documents sporring$ echo $PATH
2 /Applications/Maple
    17/:/Applications/PackageMaker.app/Contents/MacOS/:
    /Applications/MATLAB_R2014b.app/bin/:/opt/local/bin:
    /opt/local/sbin:/usr/local/bin:/usr/bin:/bin:/usr/sbin:
    /sbin:/opt/X11/bin:/Library/TeX/texbin
3 FN11194:Documents sporring$
```

The search-path can be changed by editing the setup file for Bash. On MacOS X it is called `~/.profile`, and on Linux it is either `~/.bash_profile` or `~/.bashrc`. Here new paths can be added by adding the following line: `export PATH="<new path>:<another new path>:$PATH"`.

A useful feature of Bash is that the console can help you write commands. E.g., if you write `fs` followed by pressing the `tab`-key, and if Mono is in the search-path, then Bash will typically respond by completing the line as `fsharp`, and by further pressing the `tab`-key some times, Bash will show the list of options, typically `fshpari` and `fsharpc`. Also, most commands have an extensive manual which can be accessed using the `man` command. E.g., the manual for `rm` is retrieved by `man rm`.

B | Number Systems on the Computer

B.1. Binary Numbers

Humans like to use the *decimal number* system for representing numbers. Decimal numbers are *base* 10 meaning that a decimal number consists of a sequence of digits separated by a *decimal point*, where each *digit* can have values $d \in \{0, 1, 2, \ldots, 9\}$ and the weight of each digit is proportional to its place in the sequence of digits with respect to the decimal point, i.e., the number $357.6 = 3 \cdot 10^2 + 5 \cdot 10^1 + 7 \cdot 10^0 + 6 \cdot 10^{-1}$, or in general, for a number consisting of digits d_i with $n + 1$ and m digits to the left and right of the decimal point, the value v is calculated as:

· 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 and have digits $o \in \{0, 1, \ldots, 7\}$. Octals are useful short-hand for binary, since 3 binary digits map to the set of octal digits. Likewise, *hexadecimal numbers* are base 16 with digits $h \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f\}$, such that $a_{16} = 10$, $b_{16} = 11$ and so on. Hexadecimals are convenient, since 4 binary digits map directly to the set of hexadecimal digits. Thus $367 = 101101111_2 = 557_8 = 16f_{16}$. A list of the integers 0–63 in various bases is given in Table B.1.

· 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

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.

is infinite in size both in the sense that there is no largest nor smallest number, and that between any 2 given numbers there are infinitely many numbers. Reals are widely used for calculation, but since any computer only has finite memory, there are infinitely many numbers which cannot be represent on a computer. Hence, any computation performed on a computer with reals must rely on approximations. *IEEE 754 double precision floating-point format* (*binary64*), known as a *double*, is a standard for representing an approximation of reals using 64 bits. These bits are divided into 3 parts: sign, exponent and fraction,

- IEEE 754 double precision floating-point format
- binary64
- double

$$s\, e_1 e_2 \dots e_{11}\, m_1 m_2 \dots m_{52},$$

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 exceptions that

	$m = 0$	$m \neq 0$
$e = 0$	$v = (-1)^s 0$ (signed zero)	$v = (-1)^s m 2^{1-1023}$ (subnormals)
$e = 2^{11} - 1$	$v = (-1)^s \infty$	$v = (-1)^s$ NaN (not a number)

· 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 is

$$v = (-1)^s \left(\sum_{j=1}^{52} m_j 2^{-j} \right) 2^{-1022} \tag{B.15}$$

$$= \pm \left(\sum_{j=1}^{52} m_j 2^{-j} 2^{-1022} \right) \tag{B.16}$$

$$= \pm \left(\sum_{j=1}^{52} m_j 2^{-j-1022} \right) \tag{B.17}$$

$$\overset{k=-j-1022}{=} \pm \left(\sum_{j=-1023}^{-1074} m_{-k-1022} 2^k \right), \tag{B.18}$$

which gives a distance between numbers of $2^{-1074} \simeq 10^{-323}$ in the range $0 < |v| < 2^{-1022} \simeq 10^{-308}$.

C | Commonly Used Character Sets

Letters, digits, symbols, and space are the core of how we store data, write programs, and communicate with computers and each other. These symbols are in short called characters and represent a mapping between numbers, also known as codes, and a pictorial representation of the character. E.g., the ASCII code for the letter 'A' is 65. These mappings are for short called character sets, and due to differences in natural languages and symbols used across the globe, many different character sets are in use. E.g., the English alphabet contains the letters 'a' to 'z'. These letters are common to many other European languages which in addition use even more symbols and accents. For example, Danish has further the letters 'æ', 'ø', and 'å'. Many non-European languages have completely different symbols, where the Chinese character set is probably the most extreme, and some definitions contain 106,230 different characters, albeit only 2,600 are included in the official Chinese language test at the highest level.

Presently, the most common character set used is Unicode Transformation Format (UTF), whose most popular encoding schemes are 8-bit (UTF-8) and 16-bit (UTF-16). Many other character sets exist, and many of the later build on the American Standard Code for Information Interchange (ASCII). The ISO-8859 codes were an intermediate set of character sets that are still in use, but which is greatly inferior to UTF. Here we will briefly give an overview of ASCII, ISO-8859-1 (Latin1), and UTF.

C.1. ASCII

The *American Standard Code for Information Interchange* (*ASCII*) [8], is a 7 bit code tuned for the letters of the English language, numbers, punctuation symbols, control codes and space, see Tables C.1 and C.2. The first 32 codes are reserved for 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 uppercase letter with code c may be converted to lower case by adding 32 to its code. The ASCIIbetical order also has a consequence for sorting, i.e., when sorting characters according to their ASCII code, 'A' comes before 'a', which comes before the symbol '{'.

· ASCIIbetical order

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.2. ISO/IEC 8859

The ISO/IEC 8859 report `http://www.iso.org/iso/catalogue_detail?csnumber=28245` defines 10 sets of codes specifying up to 191 codes and graphics characters using 8 bits. Set 1, also known as ISO/IEC 8859-1, Latin alphabet No. 1, or *Latin1*, covers many European languages and is designed to be compatible with ASCII, such that code for the printable characters in ASCII is the same in ISO 8859-1. Table C.3 shows the characters above 7e. Codes 00-1f and 7f-9f are undefined in ISO 8859-1.

· Latin1

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 is identical to ASCII, see Table C.1, and code points 128-255 are called the *Latin-1 Supplement block*, and are identical to the upper range of ISO 8859-1, see Table C.3. Each code-point has a number of attributes such as the *Unicode general category*. Presently more than 128,000 code points are defined as covering 135 modern and historical writing systems, and obtained 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

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.

text visualized as 'A'. More digits are used for code points of the remaining planes.

The general category is used to specify valid characters that do not necessarily have a visual representation but possibly transform text. Some categories and their letters in the first 256 code points are shown in Table C.5.

To store and retrieve code points, they must be encoded and decoded. A common encoding is *UTF-8*, which encodes code points as 1 to 4 bytes, and which is backward-compatible with ASCII and ISO 8859-1. Hence, in all 3 coding systems, the character with code 65 represents the character 'A'. Another popular encoding scheme is *UTF-16*, which encodes characters as 2 or 4 bytes, but which is not backward-compatible with ASCII or ISO 8859-1. UTF-16 is used internally in many compilers, interpreters, and operating systems.

· UTF-8

· UTF-16

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 letter
Ll	U+0061–U+007A, U+00B5, U+00DF–U+00F6, U+00F8–U+00FF	Lower case letter
Lt	None	Digraphic letter, with first part uppercase
Lm	None	Modifier letter
Lo	U+00AA, U+00BA	Gender ordinal indicator
Nl	None	Letterlike numeric character
Pc	U+005F	Low line
Mn	None	Nonspacing combining mark
Mc	None	Spacing combining mark
Cf	U+00AD	Soft Hyphen

Table C.5.: Some general categories for the first 256 code points.

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

`()`, 16
`.[]`, 35
`;;`, 18

American Standard Code for Information Interchange, 412
and, 29
ASCII, 412
ASCIIbetical order, 34, 412

base, 22, 408
`bash`, 404
Basic Latin block, 413
Basic Multilingual plane, 413
basic types, 21
binary number, 22, 408
binary operator, 27
binary64, 409
binding, 15
bit, 22, 408
blocks, 413
`bool`, 21
byte, 408
`byte[]`, 24
`byte`, 24

call-back functions, 8
`cd`, 402, 406
`char`, 21, 23
character, 23
class, 26, 35
code point, 23, 413
compile mode, 11
console, 11, 400

debugging, 14, 17
`decimal`, 24
decimal number, 22, 408
decimal point, 16, 22, 408
declarative programming, 8
`del`, 403
digit, 22, 408
`dir`, 402
directory, 400
`do`, 15
do-binding, 15
dot notation, 35
double, 409
`double`, 24
downcasting, 26

`echo`, 403, 406
encapsulate, 18
error message, 17
escape sequences, 23
event-driven programming, 8
exception, 32
exclusive or, 33
executable file, 11
`exn`, 21
expression, 8, 15, 27

`float`, 21
`float32`, 24
floating point number, 22
floating point numbers, 16
folder, 400
format string, 16
fractional part, 22, 26
function, 8, 16, 18
functional programming, 8

graphical user interface, 400
GUI, 400

hexadecimal number, 23, 408

IEEE 754 double precision floating-point format, 409
imperative programming, 8
implementation file, 11
infix notation, 27
`int`, 21
`int16`, 24
`int32`, 24
`int64`, 24
`int8`, 24

integer, 22
integer division, 31
integer number, 15
interactive mode, 11
`it`, 16, 21

keyword, 15

Latin-1 Supplement block, 413
Latin1, 413
least significant bit, 408
`let`, 15
let-binding, 15
lexeme, 18
lexical scope, 18
library file, 11
literal, 21
literal type, 24
`ls`, 405

member, 26
method, 35
`mkdir`, 402, 406
most significant bit, 408
`move`, 403
`mv`, 406

namespace, 26
NaN, 410
newline, 23
not, 29
not a number, 410

`obj`, 21
object, 8, 35
Object-oriented programming, 8
octal number, 23, 408
operands, 27
operator, 28
or, 29
overflow, 30

precedence, 27, 28
prefix operator, 27
`printfn`, 16

reals, 408
recursion, 27
redirection, 403, 406
remainder, 31
`rm`, 406
`rmdir`, 403, 406
rounding, 26
runtime error, 32

`sbyte`, 24
scientific notation, 22
script file, 11
script-fragment, 11, 18
scripts, 11
search path, 404
search-path, 406
signature file, 11
`single`, 24
source code, 11
state, 8
statement, 8
statements, 15
string, 16, 23
`string`, 21
structured programming, 8
subnormals, 410

terminal, 400
truth table, 29
type, 16, 21
type declaration, 16
type inference, 14, 16
typecasting, 26

`uint16`, 24
`uint32`, 24
`uint64`, 24
`uint8`, 24
underflow, 30
Unicode, 23
Unicode general category, 413
Unicode Standard, 413
`unit`, 16, 21
unit-testing, 14
upcasting, 26
UTF-16, 414
UTF-8, 414

`val`, 16
verbatim, 24

whitespace, 23
whole part, 22, 26
Windows command line, 400
word, 408

xor, 33