

Learning to program with F#

Jon Sparring

July 16, 2016

Contents

1	Preface	3
2	Introduction	4
I	F# basics	6
3	Executing F# code	7
3.1	Source code	7
3.2	Executing programs	7
4	Quick-start guide	9
5	Numbers, Characters, and Strings	13
5.1	Integers and Reals	17
5.2	Booleans	22
5.3	Chars and Strings	23
5.4	Mutable bindings	23
6	Functions and procedures	27
6.1	Procedures	29
7	Controlling program flow	31
7.0.1	Conditional expressions	31
7.0.2	For and while loops	32
8	Tuples, Lists, Arrays, and Sequences	35
8.1	Tuples	35
8.2	Lists	35
8.3	Arrays	35
8.3.1	1 dimensional arrays	35
8.3.2	Multidimensional Arrays	38
8.4	Sequences	40
II	Imperative programming	41
9	Exceptions	42
9.1	Exception Handling	42
10	Testing programs	43

11 Input/Output	44
11.1 Console I/O	44
11.2 File I/O	44
12 Graphical User Interfaces	46
13 The Collection	47
13.1 Mutable Collections	47
13.1.1 Mutable lists	47
13.1.2 Stacks	47
13.1.3 Queues	47
13.1.4 Sets and dictionaries	47
14 Imperative programming	48
14.1 Introduction	48
14.2 Generating random texts	48
14.2.1 0'th order statistics	48
14.2.2 1'th order statistics	50
III Declarative programming	54
15 Functional programming	55
IV Structured programming	56
16 Object-oriented programming	57
V Appendix	58
A Number systems on the computer	59
A.1 Binary numbers	59
A.2 IEEE 754 floating point standard	59
B Commonly used character sets	63
B.1 ASCII	63
B.2 ISO/IEC 8859	63
B.3 Unicode	64
C A brief introduction to Extended Backus-Naur Form	67
Bibliography	70
Index	71

Chapter 1

Preface

...¹

¹Something about why this book is written, the course, and personal notes.

Chapter 2

Introduction

Programming is the act of solving a problem by writing a program to be executed on a computer. A general method for solving problems was given by George Pólya [5] and adapted to programming is:

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

Design a plan: Good designs means that programs are faster to program easier to debug and maintain. So before you start typing a program consider things like: What are the requirements and constraints for the program? Which components should the program have? How are these components to work together? Designing often involves drawing a diagram of the program, and writing pseudo-code on paper.

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

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

Programming is a very complicated process, and the steps in Pólya's list are almost always to be performed, but the order of the steps and the number of times each step is performed varies. ¹

This book focusses on 3 fundamentally different approaches to programming:

Imperative programming, which is a type of programming that *statements* to change the program's *state*. Imperative programming emphasises *how a program shall accomplish a solution* and less on *what the solution is*. A cooking recipes is an example of the spirit of imperative programming. Almost all computer hardware is designed to execute low-level programs written in imperative style. The first major language was FORTRAN [2] which emphasized imperative style of programming.

- Imperative programming
- statements
- state

- Declarative programming

¹Should we mention core activities: Requirements, Design, Construction, Testing, Debugging, Deployment, Maintenance?

Declarative programming, which emphasises *what a program shall accomplish* but not *how*. We will consider Functional programming as a type of declarative programming. A type of programming which evaluates *functions* and avoids state changes. The program consists of *expressions* instead of statements. As a consequence, the output of functions only depends on its arguments. Functional programming has its roots in lambda calculus [1], and the first language emphasizing functional programming was Lisp [3].

- Functional programming
- functions
- expressions
- Structured programming
- Object-oriented programming
- objects

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

Most programs follows a single programming paradigm as, e.g., one of the above, but are a mix. Nevertheless, this book will treat each paradigm separately to emphasize their advantages and disadvantages.

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

Interactive and compile mode F# has an interactive and a compile mode of operation.

Indentation for scope F# uses indentation to indicate scope.

Strongly typed F# is strongly typed, reducing the number of run-time errors.

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# programs interface easily with other .Net and Mono programs through the language-independent, platform-independent bytecode called Common Intermediate Language (CIL).

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>).

Learning to program requires mastering a programming language, however most programming languages contains details that are rarely used or used in contexts far from a specific programming topic. Hence, this book takes the approach to start with an introduction to the most basic concepts of F# in Part I, followed by the 3 programming paradigms in Part II–IV while gradually expanding the introduction of F# syntax and semantics. In Part V are a number of general topics given for reference. The disadvantage of this approach is that no single part contains a reference guide to F# and F# topics are revisited and expanded across the book. For further reading please consult <http://fsharp.org>.

Part I

F# basics

Chapter 3

Executing F# code

3.1 Source code

F# is a functional first programming language that 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# 4.0 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/4.0/FSharpSpec-4.0-latest.pdf>.

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. In Mono, the interactive system is started by calling **fsharpi** from the *console*, while compilation is performed with **fsharp** and execution of the compiled code is performed using the **mono** command. The various forms of fsharp programs are identified by suffixes:

.fs An *implementation file*

.fsi A *signature file*

.fsx A *script file*

.fscript Same as **.fsx**

.exe An *executable file*

· interactive
· compiled
· console

· implementation
file
· signature file
· script file

· executable file

The implementation, signature, and script files are all typically compiled to produce an executable file, but syntactically correct code 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 *modules*.

· script-fragments
· modules

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 2 ways: interactively, where a user enters 1 or more script-fragments separated by the **;;** token, or to execute a script file treated as a single script-fragment. To illustrate the difference, consider the following program, which declares a value **a** to be the decimal value 3.0 and finally print it to the console:

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

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 **;;** token:


```

$ fsharpi

F# Interactive for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License

For help type #help;;

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

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

> #quit;;

```

The interpreter is stopped by pressing `ctrl-d` or typing `"#quit;;"`. Conversely, executing the file with the interpreter as follows,

```

$ fsharpi gettingStartedStump.fsx
3

```

Finally, compiling and executing the code is performed as,

```

$ fsharpc gettingStartedStump.fsx
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
$ mono gettingStartedStump.exe
3

```

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, then it must be retranslated as `"$fsharpi gettingStartedStump.fsx"`, but since the program has been compiled, then the compile-execute only needs to be re-executed `"$ mono gettingStartedStump.exe"`. On a Macbook Pro, with a 2.9 Ghz Intel Core i5, the time the various stages takes for this script are.

Command	Time
<code>fsharpi gettingStartedStump.fsx</code>	1.88s
<code>fsharpc gettingStartedStump.fsx</code>	1.90s
<code>mono randomTextOrder0.exe</code>	0.05s

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

¹

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* a method for debugging programs.

- type inference
- debugging
- unit-testing

¹Remember to add something about the `it` value in interactive mode.

Chapter 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,

What is the sum of 357 and 864?

we have written the following program in F#,

```
let a = 357
let b = 864
let c = a + b
printfn "%A" c
```

1221

Listing 4.1: quickStartSum.fsx - A script to add 2 numbers and print the result to the console.

In box the above, we see our program was saved as a script in a file called `quickStartSum.fsx`, and in the console we executed the program by typing the command `fsharpi quickStartSum.fsx`. The result is then printed in the console to be 1221.

To solve the program we made program consisting of several lines, where each line was a *statement*. The first statement `let a = 357` used the `let` keyword to *bind* the value 357 to the name `a`. Likewise, we bound the value 864 to the name `b`, but to the name `c` we bound the result of evaluating the *expression* `a + b`. That is, first the value `a + b` was calculated by substituting the names of `a` and `b` with their values to give the expression, `357 + 864`, then this expression was evaluated by adding the values to give, 1221, and this value was finally bound to the name `c`. The last line printed the value of `c` to the console followed by a LF (line feed, see Appendix B.1) with the `printfn` function. Here `printfn` is a function of 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 any type.

Types are a central concept in F#. In the script 4.1 we bound values of types `int` and `string` to names. The values were not *declared* to have these types, instead the types were *inferred* by F#. Had we typed these statements line by line in an interactive session, then we would have seen the inferred types:

· statement
· `let`
· keyword
· binding
· expression

· format string
· `string`

· type
· type declaration
· type inference

```

> let a = 357;;

val a : int = 357

> let b = 864;;

val b : int = 864

> let c = a + b;;

val c : int = 1221

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

```

Listing 4.2: fsharpi

The an interactive session displays the type using the `val` keyword. Since the value is also responded, then the last `printfn` statement is superfluous. However, it is ill advised to design programs to be run in an interactive session, since the scripts needs to be manually copied every time it is to be run, and since the starting state may be unclear.

· `val`

Advice!

Were we to solve a slightly different problem,

What is the sum of 357.6 and 863.4?

then we would have to use floating point arithmetic instead of integers, and the program would look like,

```

let a = 357.6
let b = 863.4
let c = a + b
printfn "%A" c

```

1221.0

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

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 which is not the same as 1221. F# is very picky about types, and generally does not allow types to be mixed. E.g., in an interactive session,

```

> let a = 357;;

val a : int = 357

> let b = 863.4;;

val b : float = 863.4

> let c = a + b;;

```

```

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

/Users/sporring/Desktop/fsharpNotes/stdin(3,13): error FS0001: The
type 'float' does not match the type 'int'

```

Listing 4.4: fsharpi

we see that binding a name to a number without a decimal point is inferred to be integer, while when binding to a number with a decimal point, then the type is inferred to be a float, and when trying to add values of integer and floating point, then we get an error.

F# is a functional first programming language, and one implication is that names are constant and cannot be changed. If attempted, then F# will return an error as, e.g.,

```

let a = 357
let a = 864

/Users/sporring/repositories/fsharpNotes/quickStartRebindError.fsx
(2,5): error FS0037: Duplicate definition of value 'a'

```

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

However, if the same was performed in an interactive session,

```

> let a = 357;;

val a : int = 357

> let a = 864;;

val a : int = 864

```

Listing 4.6: fsharpi

then apparently rebinding is legal. The difference is that the `;;` token defines a new nested *scope*. A token is a letter or a word, which the F# considers as an atomic unit. A scope is an area in a program, where a binding is valid. Scopes can be *nested*, and in F# a binding may reuse names in a nested scope, in which case the previous value is *overshadowed*. I.e., attempting the same without `;;` between the two `let` statements results in an error, e.g.,

- `;;`
- token
- scope
- nested scope
- overshadow

```

> let a = 357
- let a = 864;;

let a = 864;;
----^

/Users/sporring/Desktop/fsharpNotes/stdin(2,5): error FS0037:
Duplicate definition of value 'a'

```

Listing 4.7: fsharpi

Scopes can be visualized as nested squares as shown in Figure 4.1.

In F# *functions* are also values, and defining a function `sum` as part of the solution to the above program gives,

- function

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

(a) Illegal

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

(b) Legal

Figure 4.1: Binding of the the same name in the same scope is illegal in F# 2, but legal in a different scopes. In (a) the two bindings are in the same scope, which is illegal, while in (b) the bindings are in separate scopes by the extra `;;` token, which is legal.

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

1221

Listing 4.8: quickStartSumFct.fsx - A script to add 2 numbers using a user defined function.

Entering the function into an interactive session will illustrate the inferred type, the function `sum` has: `val sum : x:int * y:int -> int`, by which is meant that `sum` is a mapping from the set product of integers with integers into integers. 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, which will give an error, if it in a nested scope is to be used for floats,

```
> let sum x y = x + y;;

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

> let c = sum 357.6 863.4;;

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

/Users/sporring/Desktop/fsharpNotes/stdin(2,13): error FS0001: This
expression was expected to have type
int
but here has type
float
```

Listing 4.9: fsharpi

A remedy is to either define the function in the same scope as its use,

```
> let sum x y = x + y
- let c = sum 357.6 863.4;;

val sum : x:float -> y:float -> float
val c : float = 1221.0
```

Listing 4.10: fsharpi

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.

Chapter 5

Numbers, Characters, and Strings

All programs rely on processing of data, and an essential property of data is its *type*. F# contains a number of built-in types, and it is designed such that it is easy to define new types. The simplest types are called *primitive types*, and a table of some of the most commonly used primitive types are shown in Table 5.1. A *literal* is a fixed value such as "3", and F# supports *literal types* which are indicated by a suffix in most cases and as shown in the table.

A name is bound to a value by the syntax,

```
"let" [ "mutable" ] ident [ ":" type ] "=" expr [ "in" expr ]
```

That is, the *let* keyword indicates that the following is a binding of a name with an expression, and that the type may be specified with the *:* token. The binding may be mutable, which will be discussed in Section 5.4, and the binding may only be for the last expression as indicated by the *in*. The simplest example of an expression is a *literal*, i.e., a constant such as the number 3 or a function. Functions will be discussed in detail in Chapter 6. Examples of let statements with *literals*

```
> let a = 3
- let b = 4u
- let c = 5.6
- let d = 7.9f
- let e = 'A'
- let f = 'B'B
- let g = "ABC"
- let h = ();;

val a : int = 3
val b : uint32 = 4u
val c : float = 5.6
val d : float32 = 7.9000001f
val e : char = 'A'
val f : byte = 66uy
val g : string = "ABC"
val h : unit = ()
```

Listing 5.1: fsharpi

Here *a*, *b*, ..., *h* are names that we have chosen, and which by the binding operation are made equivalent to the corresponding values. Note that we did not specify the type of the name, and that F# interpreted the type from the literal form of the right-hand-side. Types matter, since the operations that can be performed on integers are quite different from those that can be performed on characters and strings. I.e.,

- type
- primitive types
- literal
- literal type
- *let*
- *:*
- *in*
- literal
- literals

Metatype	Type name	Suffix	Literal	Description
Boolean	bool	none	true	Boolean values true or false
Integer	int	none or l	3	Integer values from -2,147,483,648 to 2,147,483,647
	byte	uy	3uy	Integer values from 0 to 255
	sbyte	y	3y	Integer values from -128 to 127
	int16	s	3s	Integer values from -32768 to 32767
	uint16	us	3us	Integer values from 0 to 65535
	int32	none or l	3	Synonymous with int
	uint32	u or ul	3u	Integer values from 0 to 4,294,967,295
	int64	L	3L	Integer values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
	uint64	UL	3UL	Integer values from 0 to 18,446,744,073,709,551,615
	bigint	I	3I	Integer not limited to 64 bits
Real	float	none	3.0	64-bit IEEE 754 floating point value from $-\infty$ to ∞
	double	none	3.0	Synonymous with float
	single	F or f	3.0f	A 32-bit floating point type
	float32	F or f	3.0f	Synonymous with single
	decimal	M or m	3.0m	A floating point data type that has at least 28 significant digits
Character	char	none	'c'	Unicode character
	byte	B	'c'B	ASCII character
	string	none	"abc"	Unicode sequence of characters
	byte[]	B	"abc"B	Unicode sequence of characters
	string or byte[]	@	@"\n"	Verbatim string
None	unit	none	()	No value denoted

Table 5.1: List of primitive types and the corresponding literal. The most commonly used types are highlighted in bold. Note that string verbatim uses a prefix instead of suffix notation. For at description of floating point numbers see Appendix A.2 and for ASCII and Unicode characters see Appendix B.

```

> let a = 3
- let b = 3.0
- let c = '3'
- let d = "3";;

val a : int = 3
val b : float = 3.0
val c : char = '3'
val d : string = "3"

```

Listing 5.2: fsharpi

the variables `a`, `b`, `c`, and `d` all represent the number 3, but their types are different, and hence they are quite different values. When specifying the type, then the type and the literal form must match, i.e., mixing types and literals gives an error,

```

> let a : float = 3;;

let a : float = 3;;
-----^

/Users/sporring/repositories/fsharpNotes/stdin(50,17): error FS0001
: This expression was expected to have type
float
but here has type
int

```

Listing 5.3: fsharpi

since the left-hand-side is a name of type `float` while the right-hand-side is a literal of type `integer`. Many primitive types are compatible and may be changed to each other by *type casting*. E.g.,

· type casting

```

> let a = float 3;;

val a : float = 3.0

```

Listing 5.4: fsharpi

where the left-hand-side is inferred to be of type `float`, since the integer number 3 is casted to float resulting in a similar floating point value, in this case the float point number 3.0. As a particular note, the boolean values are often treated as the integer values 0 and 1, however casting can only be performed with built-in functions, e.g.,

```

> let a = System.Convert.ToBoolean 1
- let b = System.Convert.ToBoolean 0
- let c = System.Convert.ToInt32 true
- let d = System.Convert.ToInt32 false;;

val a : bool = true
val b : bool = false
val c : int = 1
val d : int = 0

```

Listing 5.5: fsharpi

Here `System.Convert.ToBoolean` is the name of a function `ToBoolean`, which is a *member* of the

· member

`class Convert` that is included in the *namespace System*. Namespaces, classes, and members are all part of Structured programming to be discussed in Part IV. For more on functions see Section 6 Typecasting is often a destructive operation, e.g., typecasting a float to int removes the part after the decimal point without rounding,

```
let a = 357.6
let b = int a
printfn "%A -> %A" a b
```

```
357.6 -> 357
```

Listing 5.6: quickStartDownCast.fsx - Fractional part is removed by downcasting.

Here we typecasted to a lesser type, in the sense that integers is a subset of floats, which is called *downcasting*. The opposite is called *upcasting* is often non-destructive, as Listing 5.4 showed, where an integer was casted to a float while retaining its value. As a side note, *rounding* a number $y.x$, where y is the whole part and x is the fractional part, is the operation of mapping numbers in the interval $y.x \in [y.0, y.5)$ to y and $y.x \in [y.5, y + 1)$ to $y + 1$. This can be performed by downcasting as follows,

```
let a = 357.6
let b = int (a + 0.5)
printfn "%A -> %A" a b
```

```
357.6 -> 358
```

Listing 5.7: rounding.fsx - The rounding function may be obtained by downcasting.

since if $y.x \in [y.0, y.5)$, then $y.x + 0.5 \in [y.5, y + 1)$, from which downcasting removes the fractional part resulting in y . And if $y.x \in [y.5, y + 1)$, then $y.x + 0.5 \in [y + 1, y + 1.5)$, from which downcasting removes the fractional part resulting in $y + 1$. Hence, the result is rounding.

If parentheses are omitted in Listing 5.7, then F# will interpret the expression as `(int a) + 0.5`, which is erroneous, since addition of an integer with a float is undefined. This is an example of precedence, i.e., typecasting takes precedence over addition meaning that it is performed before addition. Consider the arithmetic expression, whose result is bound to `a` by

```
> let a = 3 + 4 * 5;;

val a : int = 23
```

Listing 5.8: fsharp

Here, the addition and multiplication functions are shown in *infix notation* with the *operator* tokens `+` 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 precedences, and for some common built-in operators shown in Table 5.2, the precedence is shown by the order they are given in the table. Associativity implies the order in which calculations are performed for operators of same precedence. For some operators and type combinations association matters little, e.g., multiplication associates to the left and exponentiation associates to the right, e.g., in

```
> let a = 3.0*4.0*5.0
- let b = (3.0*4.0)*5.0
```

· class
· namespace

· downcasting
· upcasting
· rounding

· infix notation
· operator

· precedence

Operator	Associativity	Example	Description
<code>f x</code>	Left	<code>f 3</code>	Function evaluation
<code>+op, -op</code>	Left	<code>-3</code>	Unary operator
<code>&&</code>	Left	<code>true && true</code>	Boolean and
<code> </code>	Left	<code>true true</code>	Boolean or
<code>op ** op</code>	Right	<code>3.0 ** 2.0</code>	Exponent
<code>op * op, op / op, op % op</code>	Left	<code>3.0 / 2.0</code>	Multiplication, division and remainder
<code>op + op, op - op</code>	Left	<code>3.0 + 2.0</code>	Addition and subtraction binary operators

Table 5.2: Some common operators, their precedence, and their associativity. Rows are ordered from highest to lowest precedences, such that `op * op` has higher precedence than `op + op`. Operators in the same row has same precedence.

```
- let c = 3.0*(4.0*5.0);;

val a : float = 60.0
val b : float = 60.0
val c : float = 60.0

> let d = 4.0 ** 3.0 ** 2.0
- let e = (4.0 ** 3.0) ** 2.0
- let f = 4.0 ** (3.0 ** 2.0);;

val d : float = 262144.0
val e : float = 4096.0
val f : float = 262144.0
```

Listing 5.9: fsharp

the expression for `a` is interpreted as `b` but gives the same results as `c` since association does not matter for multiplication of numbers, but the expression for `d` is interpreted as `f` which is quite different from `e`.

A less common notation is to define bindings for expressions using the `in` keyword, e.g.,

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

9.0

Listing 5.10: numbersIn.fsx - The name `p` is only bound in the nested scope following the keyword `in`.

Here `p` is only bound in the *scope* of the expression following the `in` keyword, in this the `printfn` statement, and `p` is unbound in lines that follows.

5.1 Integers and Reals

The set of integers and reals are infinitely large, and since all computers have limited resources, it is not possible to represent these sets in their entirety. The various integer and floating point types listed in Table 5.1 are finite subset where the integer types have been reduced by limiting their ranges and the floating point types have been reduced by sampling the space of reals. An in-depth description of integer and floating point implementations can be found in Appendix A. The `int` and `float` are the most common types.

For integers the following arithmetic operators are defined:

+op, -op: These are unary plus and minus operators, and plus has no effect, but minus changes the sign, e.g.,

```
> let a = 5
- let b = -a;;

val a : int = 5
val b : int = -5
```

Listing 5.11: fsharp

op + op, op - op, op * op: These are binary operators, where addition, subtraction and multiplication performs the usual operations,

```
> let a = 7 + 3
- let b = 7 - 3
- let c = 7 * 3;;

val a : int = 10
val b : int = 4
val c : int = 21
```

Listing 5.12: fsharp

op / op, op % op: These are binary operators, and division performs integer division, where the fractional part is discarded after division, and the `\%` is the remainder operator, which calculates the remainder after integer division,

```
> let a = 7 / 3
- let b = 7 % 3;;

val a : int = 2
val b : int = 1
```

Listing 5.13: fsharp

Together integer division and remainder is a lossless representation of the original number as,

```
> let x = 7
- let whole = x / 3
- let remainder = x % 3
- let y = whole * 3 + remainder;;

val x : int = 7
val whole : int = 2
val remainder : int = 1
val y : int = 7
```

Listing 5.14: fsharp

And we see that `x` and `y` is bound to the same value.

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

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

Listing 5.15: fsharp

which is equal to 2^5 . Note that when no `let` statement is used in conjunction with an expression then F# automatically binds the result to the `it` name, i.e., the above is equal to

```
> let it = pown 2 5;;

val it : int = 32
```

Listing 5.16: fsharp

Rumor has it, that the name `it` is an abbreviation for 'irrelevant'.

Performing arithmetic operations on `int` types requires extra care, since the result may cause *overflow*, *underflow*, and even exceptions, e.g., the range of the integer type `sbyte` is $[-128 \dots 127]$, which causes problems in the following example,

```
> let a = 100y - let b = 30y - let c = a+b;;

val a : sbyte = 100y
val b : sbyte = 30y
val c : sbyte = -126y
```

Listing 5.17: fsharp

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`,

```
> let a = -100y
- let b = -30y
- let c = a+b;;

val a : sbyte = -100y
val b : sbyte = -30y
val c : sbyte = 126y
```

Listing 5.18: fsharp

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*,

```
> 3/0;;
System.DivideByZeroException: Attempted to divide by zero.
  at <StartupCode$FSI_0007>.$FSI_0007.main@ () <0x6b78180 + 0x0000e>
  > in <filename unknown>: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,
    BindingFlags invokeAttr, System.Reflection.Binder binder,
    System.Object[] parameters, System.Globalization.CultureInfo
    culture) <0x1a55ba0 + 0x000a1> in <filename unknown>:0
```

Stopped due to error

Listing 5.19: fsharp

The output looks daunting at first sight, but the first and last line of the error message are the most important parts, which tells us what exception was cast and why the program stopped. The middle are technical details concerning which part of the program caused this, and can be ignored for the time being. Exceptions are a type of *run-time error*, and are treated in Chapter 9

· run-time error

Integers can also be written in binary, octal, or hexadecimal format using the prefixes 0b, 0o, and 0x, e.g.,

```
> let a = 0b1011
- let b = 0o13
- let c = 0xb;;

val a : int = 11
val b : int = 11
val c : int = 11
```

Listing 5.20: fsharp

For a description of binary representations see Appendix A.1. The overflow error in Listing 5.17 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`,

```
> let a = 0b10000010uy
- let b = 0b10000010y;;

val a : byte = 130uy
val b : sbyte = -126y
```

Listing 5.21: fsharp

That is, for signed bytes, the left-most bit is used to represent the sign, and since the addition of $100 = 01100100_2$ and $30 = 00011110_2$ 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`.

For floating point numbers the following arithmetic operators are defined:

+op, -op: These are unary plus and minus operators, and plus has no effect, but minus changes the sign, e.g.,

```
> let a = 5.0
- let b = -a;;

val a : float = 5.0
val b : float = -5.0
```

Listing 5.22: fsharp

op + op, op - op, op * op, op / op: These are binary operators, where addition, subtraction, multiplication, and division performs the usual operations,

```
> let a = 7.0 + 3.0
- let b = 7.0 - 3.0
- let c = 7.0 * 3.0
- let d = 7.0 / 3.0;;
```

```

val a : float = 10.0
val b : float = 4.0
val c : float = 21.0
val d : float = 2.333333333

```

Listing 5.23: fsharp

`op %` `op`: The binary remainder operator, and division performs integer division, where the fractional part is discarded after division, and the `\%` is the remainder operator, which calculates the remainder after integer division,

```

> let a = 7.0 / 3.0
- let b = 7.0 % 3.0;;

val a : int = 2.0
val b : int = 1.0

```

Listing 5.24: fsharp

The remainder for floating point numbers can be fractional, but division, rounding, and remainder is still a lossless representation of the original number as,

```

> let x = 7.0
- let division = x / 3.2
- let whole = float (int (division + 0.5))
- let remainder = x % 3.2
- let y = whole * 3.2 + remainder;;

val x : float = 7.0
val division : float = 2.1875
val whole : float = 2.0
val remainder : float = 0.6
val y : float = 7.0

```

Listing 5.25: fsharp

And we see that `x` and `y` is bound to the same value.

`op **` `op`: In spite of an unusual notation, the binary exponentiation operator performs the usual calculation,

```

> let a = 2.0 ** 5.0;;

val a : float = 32.0

```

Listing 5.26: fsharp

which is equal to 2^5 .

Arithmetic using `float` will not cause over- and underflow problems, since the IEEE 754 standard includes the special numbers $\pm\infty$ and NaN. E.g.,

```

> let a = 1.0/0.0
- let b = 0.0/0.0;;

val a : float = infinity

```

a	b	$a \cdot b$	$a + b$	\bar{a}
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Table 5.3: Truth table for boolean 'and', 'or', and 'not' operators. Value 0 is false and 1 is true.

```
val b : float = nan
```

Listing 5.27: fsharp

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.,

```
> let a = 357.8
- let b = a+0.1
- let c = b+0.1
- let d = c - 358.0;;

val a : float = 357.8
val b : float = 357.9
val c : float = 358.0
val d : float = 5.684341886e-14
```

Listing 5.28: fsharp

Hence, although `c` appears to be correctly calculated, by the subtraction we see, that the value bound in `c` is not exactly the same as 358.0, and the reason is that neither 357.8 nor 0.1 are exactly representable as a `float`, which is why the repeated addition accumulates a small representation error.

5.2 Booleans

Boolean arithmetic is the basis of almost all computers and particularly important for controlling program flow, which will be discussed in Chapter 7. Boolean values are one of 2 possible values, true or false, which is also sometimes written as 1 and 0. Two basic operations on boolean values are 'and' often also written as multiplication, and 'or' often written as addition, and 'not' often written as a bar above the value. All possible combination of input on these values can be written on tabular form, known as a *truth table*, shown in Table 5.3. That is, the multiplication and addition are good mnemonics for remembering the result of the 'and' and 'or' operators. In F# the values `true` and `false` are used, and the operators `&&` for 'and', `||` for 'or', and the function `not` for 'not', such that the above table is reproduced by,

```
> let t = true
- let f = false
- printfn "a      b      a*b    a+b    not a"
- printfn "%A %A %A %A %A" f f (f && f) (f || f) (not f)
- printfn "%A %A %A %A %A" f t (f && t) (f || t) (not f)
- printfn "%A %A %A %A %A" t f (t && f) (t || f) (not t)
- printfn "%A %A %A %A %A" t t (t && t) (t || t) (not t);;
a      b      a*b    a+b    not a
false false false false true
false true  false true  true
true  false false true  false
```

· and
· or
· not
· truth table

```

true   true   true   true   false

val t  : bool = true
val f  : bool = false
val it : unit = ()

```

Listing 5.29: fsharp

Careful spacing in the format string of the `printfn` function was used to align columns. Next section will discuss more elegant formatting options.

5.3 Chars and Strings

...

5.4 Mutable bindings

The `mutable` in `let` bindings means that the name may be rebound to a new value using the following syntax,

```
ident "<-" expr
```

Mutable data is synonymous with the term *variable*. A variable is an area in the computer's working memory associated with a name and a type, and this area may be read from and written to during program execution. For example,

· Mutable data
· variable

```

let mutable x = Unchecked.defaultof<int> // Declare a variable x of
    type int and assign the corresponding default value to it.
printfn "%d" x
x <- 5 // Assign a new value 5 to x
printfn "%d" x
x <- -3 // Assign a new value -3 to x
printfn "%d" x

```

```

0
5
-3

```

Listing 5.30: mutableAssignReassign.fsx -

Here an area in memory was denoted `x`, declared as type integer and assigned a default value. Later, this value of `x` was replaced with another integer and yet another integer. The operator '`<-`' is used to distinguish the statement from the mathematical concept of equality. A short-hand for the above is available as,

```

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

```

```

5

```


-3

Listing 5.31: mutableAssignReassingShort.fsx -

where the assignment of the default value was skipped, and the type was inferred from the assignment operation. However, it's important to note, that when the variable is declared, then the ' $|=$ ' operator must be used, while later reassignments must use the ' $|<=$ ' operator. Type mismatches will result in an error,

```
let mutable x = 5 // Declare a variable x and assign the value 5 to
    it
printfn "%d" x
x <- -3.0 // This is illegal, since -3.0 is a float, while x is of
    type int
printfn "%d" x
```

```
/Users/sporring/repositories/fsharpNotes/
    mutableAssignReassingTypeError.fsx(3,6): error FS0001: This
    expression was expected to have type
        int
but here has type
    float
```

Listing 5.32: mutableAssignReassingTypeError.fsx -

A typical variable is a counter of type integer, and a typical use of counters is to increment them, i.e., erasing a new value to be one more than its previous value. For example,

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

5
6

Listing 5.33: mutableAssignIncrement.fsx -

An function that elegantly implements the incrementation operation may be constructed as,

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

1
2

3

Listing 5.34: mutableAssignIncrementEncapsulation.fsx -

¹ Here the output of `incr` is an anonymous function, that takes no argument, increments the variable of `incr` and returns the new value of the counter. This construction is called *encapsulation*, since the variable `counter` is hidden by the function `incr` from the user, i.e., the user need not be concerned with how the increment operator is implemented and the variable name used by `incr` does not clutter the scope where it is used.

· encapsulation

Variables cannot be returned from functions, that is,

```
let g () =  
    let x = 0  
    x  
printfn "%d" (g ())
```

0

Listing 5.35: mutableAssignReturnValue.fsx -

declares a function that has no arguments and returns the value 0, while the same for a variable is illegal,

```
let g () =  
    let mutual x = 0  
    x  
printfn "%d" (g ())
```

```
/Users/sporring/repositories/fsharpNotes/  
mutableAssignReturnVariable.fsx(3,3): error FS0039: The value  
or constructor 'x' is not defined
```

Listing 5.36: mutableAssignReturnVariable.fsx -

There is a workaround for this by using *reference cells* by the build-in function `ref` and operators `!|` and `|:=|`,

· reference cells

```
let g () =  
    let x = ref 0  
    x  
let y = g ()  
printfn "%d" !y  
y := 3  
printfn "%d" !y
```

0

3

Listing 5.37: mutableAssignReturnRefCell.fsx -

That is, the `ref` function creates a reference variable, the `'!|'` and the `'|:=|'` operators reads and writes its value. Reference cells are in some language called pointers, and their use is strongly discouraged,

since they may cause *side-effects*, which is the effect that one function changes the state of another, · side-effects such as the following example demonstrates,

```
let updateFactor factor =  
    factor := 2  
  
let multiplyWithFactor x =  
    let a = ref 1  
    updateFactor a  
    !a * x  
  
printfn "%d" (multiplyWithFactor 3)
```

6

Listing 5.38: mutableAssignReturnSideEffect.fsx -

In the example, the function `updateFactor` changes a variable in the scope of `multiplyWithFactor`, which is prone to errors, since the style of programming does not follow the usual assignment syntax. Better style of programming is,

```
let updateFactor () =  
    2  
  
let multiplyWithFactor x =  
    let a = ref 1  
    a := updateFactor ()  
    !a * x  
  
printfn "%d" (multiplyWithFactor 3)
```

6

Listing 5.39: mutableAssignReturnWithoutSideEffect.fsx -

Here there can be no doubt in `multiplyWithFactor` that the value of 'a' is changing. Side-effects do have their use, but should in general be avoided at almost all costs, and in general it is advised to refrain from using ref cells.

¹Explain why this works!

Chapter 6

Functions and procedures

to define a function, we use the syntax

```
"let" name arg { arg } "=" expr
```

where **arg** is a list of argument names, and **expr** is an expression. D
or specify the type of the function at point of definition using the notation,

```
"let" name argWType { argWType } [ ":" type ] "=" expr  
argWType = arg | "(" arg ":" type ")"
```

where not all types need to be declared, just sufficient for F# to be able to infer the types for the full statement. In the example, one sufficient specification is,

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

Listing 6.1: fsharpi

but alternatively we could have specified the type of the result,

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

or even just one of the arguments,

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

In both cases, since the `+` *operator* is only defined for *operands* of the same type, then when the type of either the result, any or both operands are declared, then the type of the remaining follows directly. A function is a mapping between an input and output domain. F# is a functional first programming language, and offers a number of alternative methods for specifying parameters. A simple example is,

```
let mul (x, y) = x*y  
let z = mul (3, 5)  
printfn "%d" z
```

· operator
· operand

15

Listing 6.2: functionDeclarationMul.fsx -

which declares a function of a tuple and returns their multiplication. The types are inferred from its first use in the second line, i.e., `mul` is `val mul : x:int * y:int -> int`. An argument may be of generic type for input, which need not be inferred without sacrificing type safety, e.g.,

```
let second (x, y) = y
let a = second (3, 5)
printfn "%A" a
let b = second ("horse", 5.0)
printfn "%A" b
```

5

5.0

Listing 6.3: functionDeclarationGeneric.fsx -

Here the function `second` does not use the first element in the tuple, `x`, and the type of the second element, `y`, can safely be anything.

Functions may be anonymously declared using the `fun` keyword,

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

5

Listing 6.4: functionDeclarationAnonymous.fsx -

Anonymous functions are often used as arguments to other functions, e.g.,

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

18

Listing 6.5: functionDeclarationAnonymousAdvanced.fsx -

This is a powerful concept, but can make programs hard to read, and overly use is not recommended. Functions may be declared using pattern matching, which is a flexible method for declaring output depending on conditions on the input value. The most common pattern matching method is by use of the `match with` syntax,

```
let rec factorial n =
    match n with
    | 0 -> 1
    | 1 -> 1
    | _ -> n * (factorial (n - 1))

printfn "%d" (factorial 5)
```

```
120
```

Listing 6.6: functionDeclarationMatchWith.fsx -

A short-hand only for functions of 1 parameter is the `function` syntax,

```
let rec factorial = function
| 0 -> 1
| 1 -> 1
| n -> n * (factorial (n - 1))

printfn "%d" (factorial 5)
```

```
120
```

Listing 6.7: functionDeclarationFunction.fsx -

Note that the name given in the match, here `n`, is not used in the first line, and is arbitrary at the line of pattern matchin, and may even be different on each line. For these reasons is this syntax discouraged. Functions may be declared from other functions

```
let mul (x, y) = x*y
let double y = mul (2.0, y)
printfn "%g" (mul (5.0, 3.0))
printfn "%g" (double 3.0)
```

```
15
```

```
6
```

Listing 6.8: functionDeclarationTupleCurrying.fsx -

For functions of more than 1 argument, there exists a short notation, which is called *currying* in tribute of Haskell Curry, · currying

```
let mul x y = x*y
let double = mul 2.0
printfn "%g" (mul 5.0 3.0)
printfn "%g" (double 3.0)
```

```
15
```

```
6
```

Listing 6.9: functionDeclarationCurrying.fsx -

Here `mul 2.0` is a partial specification of the function `mul x y`, where the first argument is fixed, and hence, `double` is a function of 1 argument being the second argument of `mul`. Currying is often used in functional programming, but generally currying should be used carefully, since currying may seriously reduce readability of code.

6.1 Procedures

A *procedure* is a generalisation of the concept of functions, and in contrast to functions procedures need not return values. An example, we've already seen is the `printfn`, which is used to print text on the console, but does not return a value. Coincidentally, since the console is a state, printing to it is a side-effect. Above we examined · procedure

```
let updateFactor factor =  
  factor := 2
```

which also does not have a return value. Procedural thinking is useful for encapsulation, but is prone to side-effects and should be minimized by being replaced by functional thinking.

Chapter 7

Controlling program flow

7.0.1 Conditional expressions

```
"if" expr "then" expr  
[{"elif" expr "then" expr}  
"else" expr]
```

A basic flow control mechanism used both for functional and imperative programming is the **if-then-else** construction, e.g.,

```
let printOnlyPostiveValues x =  
  if x > 0 then  
    printfn "%d" x  
printOnlyPostiveValues 3  
printOnlyPostiveValues -3
```

3

Listing 7.1: flowIfThen.fsx -

I.e., if and only if the value of the argument is postive, then it will be printed on screen. More common is to include the **else**

```
let abs x =  
  if x < 0 then  
    -x  
  else  
    x  
printfn "%d" (abs 3)  
printfn "%d" (abs -3)
```

3

3

Listing 7.2: flowIfThenElse.fsx -

A common construction is a nested list of **if-then-else**,

```
let digitToString x =  
  if x < 1 then
```



```

    '0'
  else
    if x < 2 then
      '1'
    else
      '2'

printfn "%c" (digitToString 1)
printfn "%c" (digitToString 3)
printfn "%c" (digitToString -3)

```

```

1
2
0

```

Listing 7.3: flowIfThenElseNested.fsx -

where the integers 0-2 are converted to characters, and integers outside this domain is converted to the nearest equivalent number. This construction is so common that a short-hand notation exists, and we may equivalently have written,

```

let digitToString x =
  if x < 1 then
    '0'
  elif x < 2 then
    '1'
  else
    '2'

printfn "%c" (digitToString 1)
printfn "%c" (digitToString 3)
printfn "%c" (digitToString -3)

```

```

1
2
0

```

Listing 7.4: flowIfThenElseNestedShort.fsx -

7.0.2 For and while loops

A major difference between functional and imperative programming is how loops are expressed. Consider the problem of printing the numbers 1 to 5 on the console with a `while` loop can be done as follows,

```

let mutable i = 1
while i <= 5 do
  printf "%d " i
  i <- i + 1
printf "\n"

```

```

1 2 3 4 5

```

Listing 7.5: flowWhile.fsx -

where the same result by recursion as

```
let rec prt a b =  
  if a <= b then  
    printf "%d " a  
    prt (a + 1) b  
  else  
    printf "\n"  
prt 1 5
```

1 2 3 4 5

Listing 7.6: flowWhileRecursion.fsx -

The counting example is so often used that a special notation is available, the `for` loop, where the above could be implemented as

```
for i = 1 to 5 do  
  printf "%d " i  
printf "\n"
```

1 2 3 4 5

Listing 7.7: flowFor.fsx -

Note that `i` is a value and not a variable here. For a more complicated example, consider the problem of calculating average grades from a list of courses and grades. Using the above construction, this could be performed as,

```
let courseGrades =  
  ["Introduction to programming", 95;  
   "Linear algebra", 80;  
   "User Interaction", 85;]  
  
let mutable sum = 0;  
let mutable n = 0;  
for i = 0 to (List.length courseGrades) - 1 do  
  let (title, grade) = courseGrades.[i]  
  printfn "Course: %s, Grade: %d" title grade  
  sum <- sum + grade;  
  n <- n + 1;  
let avg = (float sum) / (float n)  
printfn "Average grade: %g" avg
```

Course: Introduction to programming, Grade: 95
Course: Linear algebra, Grade: 80
Course: User Interaction, Grade: 85
Average grade: 86.6667

Listing 7.8: flowForListsIndex.fsx -

However, an elegant alternative is available as

```

let courseGrades =
  ["Introduction to programming", 95;
   "Linear algebra", 80;
   "User Interaction", 85;]

let mutable sum = 0;
let mutable n = 0;
for (title, grade) in courseGrades do
  printfn "Course: %s, Grade: %d" title grade
  sum <- sum + grade;
  n <- n + 1;
let avg = (float sum) / (float n)
printfn "Average grade: %g" avg

```

```

Course: Introduction to programming, Grade: 95
Course: Linear algebra, Grade: 80
Course: User Interaction, Grade: 85
Average grade: 86.6667

```

Listing 7.9: flowForLists.fsx -

This to be preferred, since we completely can ignore list boundary conditions and hence avoid out of range indexing. For comparison see a recursive implementation of the same,

```

let courseGrades =
  ["Introduction to programming", 95;
   "Linear algebra", 80;
   "User Interaction", 85;]

let rec printAndSum lst =
  match lst with
  | (title, grade)::rest ->
    printfn "Course: %s, Grade: %d" title grade
    let (sum, n) = printAndSum rest
    (sum + grade, n + 1)
  | _ -> (0, 0)
let (sum, n) = printAndSum courseGrades
let avg = (float sum) / (float n)
printfn "Average grade: %g" avg

```

```

Course: Introduction to programming, Grade: 95
Course: Linear algebra, Grade: 80
Course: User Interaction, Grade: 85
Average grade: 86.6667

```

Listing 7.10: flowForListsRecursive.fsx -

Note how this implementation avoids the use of variables in contrast to the previous examples.

Chapter 8

Tuples, Lists, Arrays, and Sequences

8.1 Tuples

8.2 Lists

8.3 Arrays

8.3.1 1 dimensional arrays

Roughly speaking, arrays are mutable lists, and may be created and indexed in the same manner, e.g.,

```
let A = [| 1; 2; 3; 4; 5 |]
let B = [| 1 .. 5 |]
let C = [| for a in 1 ..5 do yield a |]

let printArray (a : int array) =
    for i = 0 to a.Length - 1 do
        printf "%d " a.[i]
    printf "\n"

printArray A
printArray B
printArray C
```

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Listing 8.1: arrayCreation.fsx -

Notice that as for lists, arrays are indexed starting with 0, and that in this particular case it was necessary to specify the type of the argument for `printArray` as an array of integers with the `array` keyword. The `array` keyword is synonymous with `'[]'`. Arrays do not support pattern matching, cannot be resized, but are mutable,

```
let A = [| 1; 2; 3; 4; 5 |]

let printArray (a : int array) =
```

```

    for i = 0 to a.Length - 1 do
        printf "%d " a.[i]
    printf "\n"

let square (a : int array) =
    for i = 0 to a.Length - 1 do
        a.[i] <- a.[i] * a.[i]

printArray A
square A
printArray A

```

```

1 2 3 4 5
1 4 9 16 25

```

Listing 8.2: arrayReassign.fsx -

Notice that in spite the missing `mutable` keyword, the function `square` still had the side-effect of squaring alle entries in `A`. Arrays support *slicing*, that is, indexing an array with a range results in a copy of array with values corresponding to the range, e.g.,

```

let A = [| 1; 2; 3; 4; 5 |]
let B = A.[1..3]
let C = A[..2]
let D = A.[3..]
let E = A.[*]

let printArray (a : int array) =
    for i = 0 to a.Length - 1 do
        printf "%d " a.[i]
    printf "\n"

printArray A
printArray B
printArray C
printArray D
printArray E

```

```

1 2 3 4 5
2 3 4
1 2 3
4 5
1 2 3 4 5

```

Listing 8.3: arraySlicing.fsx -

As illustrated, the missing start or end index implies from the first or to the last element. There are quite a number of built-in procedures for all arrays some of which we summarize in Table 8.1. Thus, the `arrayReassign.fsx` program can be written using arrays as,

```

let A = [| 1 .. 5 |]

let printArray (a : int array) =
    Array.iter (fun x -> printf "%d " x) a

```

append	Creates an array that contains the elements of one array followed by the elements of another array.
average	Returns the average of the elements in an array.
blit	Reads a range of elements from one array and writes them into another.
choose	Applies a supplied function to each element of an array. Returns an array that contains the results <i>x</i> for each element for which the function returns <i>Some(x)</i> .
collect	Applies the supplied function to each element of an array, concatenates the results, and returns the combined array.
concat	Creates an array that contains the elements of each of the supplied sequence of arrays.
copy	Creates an array that contains the elements of the supplied array.
create	Creates an array whose elements are all initially the supplied value.
empty	Returns an empty array of the given type.
exists	Tests whether any element of an array satisfies the supplied predicate.
fill	Fills a range of elements of an array with the supplied value.
filter	Returns a collection that contains only the elements of the supplied array for which the supplied condition returns true.
find	Returns the first element for which the supplied function returns true. Raises <i>System.Collections.Generic.KeyNotFoundException</i> if no such element exists.
findIndex	Returns the index of the first element in an array that satisfies the supplied condition. Raises <i>System.Collections.Generic.KeyNotFoundException</i> if none of the elements satisfy the condition.
fold	Applies a function to each element of an array, threading an accumulator argument through the computation. If the input function is <i>f</i> and the array elements are <i>i0...iN</i> , this function computes <i>f (...(f s i0)...) iN</i> .
foldBack	Applies a function to each element of an array, threading an accumulator argument through the computation. If the input function is <i>f</i> and the array elements are <i>i0...iN</i> , this function computes <i>f i0 (...(f iN s))</i> .
forall	Tests whether all elements of an array satisfy the supplied condition.
isEmpty	Tests whether an array has any elements.
iter	Applies the supplied function to each element of an array.
init	...
length	Returns the length of an array. The <i>System.Array.Length</i> property does the same thing.
map	Creates an array whose elements are the results of applying the supplied function to each of the elements of a supplied array.
mapI	
max	Returns the largest of all elements of an array. <i>Operators.max</i> is used to compare the elements.
min	Returns the smallest of all elements of an array. <i>Operators.min</i> is used to compare the elements.
ofList	Creates an array from the supplied list.
ofSeq	Creates an array from the supplied enumerable object.
partition	Splits an array into two arrays, one containing the elements for which the supplied condition returns true, and the other containing those for which it returns false.
rev	Reverses the order of the elements in a supplied array.
sort	Sorts the elements of an array and returns a new array. <i>Operators.compare</i> is used to compare the elements.
sub	Creates an array that contains the supplied subrange, which is specified by starting index and length.
sum	Returns the sum of the elements in the array.
toList	Converts the supplied array to a list.
toSeq	Views the supplied array as a sequence.
unzip	Splits an array of tuple pairs into a tuple of two arrays.
zeroCreate	Creates an array whose elements are all initially zero.
zip	Combines two arrays into an array of tuples that have two elements. The two arrays must have equal lengths; otherwise, <i>System.ArgumentException</i> is raised.

```

    printf "\n"

let square a = a * a

printArray A
let B = Array.map square A
printArray A
printArray B

```

```

1 2 3 4 5
1 2 3 4 5
1 4 9 16 25

```

Listing 8.4: arrayReassignModule.fsx -

and the flowForListsIndex.fsx program can be written using arrays as,

```

let courseGrades =
    ["Introduction to programming", 95;
     "Linear algebra", 80;
     "User Interaction", 85;]

let A = Array.ofList courseGrades
let printCourseNGrade (title, grade) =
    printfn "Course: %s, Grade: %d" title grade
Array.iter printCourseNGrade A
let (titles,grades) = Array.unzip A
let avg = (float (Array.sum grades)) / (float grades.Length)
printfn "Average grade: %g" avg

```

```

Course: Introduction to programming, Grade: 95
Course: Linear algebra, Grade: 80
Course: User Interaction, Grade: 85
Average grade: 86.6667

```

Listing 8.5: flowForListsIndexModule.fsx -

Both cases avoid the use of variables and side-effects which is a big advantage for code safety.

8.3.2 Multidimensional Arrays

Higher dimensional arrays can be created as arrays of arrays (of arrays ...). These are known as *jagged arrays*, since there is no inherent control of that all sub-arrays are of similar size. E.g., the following is a jagged array of increasing width, · jagged arrays

```

let A = [| for n in 1..3 do yield [| 1 .. n |] |]

let printArrayOfArrays (a : int array array) =
    for i = 0 to a.Length - 1 do
        for j = 0 to a.[i].Length - 1 do
            printf "%d " a.[i].[j]
        printf "\n"

printArrayOfArrays A

```

```
1
1 2
1 2 3
```

Listing 8.6: arrayJagged.fsx -

Indexing arrays of arrays is done sequentially, in the sense that in the above example, the number of outer arrays is `a.Length`, `a.[i]` is the *i*'th array, the length of the *i*'th array is `a.[i].Length`, and the *j*'th element of the *i*'th array is thus `a.[i].[j]`. Often 2 dimensional square arrays are used, which can be implemented as a jagged array as,

```
let pownArray (a : int array) p =
    for i = 0 to a.Length - 1 do
        a.[i] <- pown a.[i] p
    a

let A = [| for n in 1..3 do yield (pownArray [| 1 .. 4 |] n ) |]

let printArrayOfArrays (a : int array array) =
    for i = 0 to a.Length - 1 do
        for j = 0 to a.[i].Length - 1 do
            printf "%2d " a.[i].[j]
        printf "\n"

printArrayOfArrays A
```

```
1  2  3  4
1  4  9 16
1  8 27 64
```

Listing 8.7: arrayJaggedSquare.fsx -

In fact, square arrays of dimensions 2 to 4 are so common that fsharp has built-in modules for their support. In the following describe `Array2D`. The workings of `Array3D` and `Array4D` are very similar. An example of creating the same 2 dimensional array as above but as an `Array2D` is,

```
let A = Array2D.create 3 4 0
for i = 0 to (Array2D.length1 A) - 1 do
    for j = 0 to (Array2D.length2 A) - 1 do
        A.[i,j] <- pown (j + 1) (i + 1)

let printArray2D (a : int [,]) =
    for i = 0 to (Array2D.length1 a) - 1 do
        for j = 0 to (Array2D.length2 a) - 1 do
            printf "%2d " a.[i, j]
        printf "\n"

printArray2D A
```

```
1  2  3  4
1  4  9 16
1  8 27 64
```

Listing 8.8: array2D.fsx -

blit	Reads a range of elements from one array and writes them into another.
copy	Creates an array that contains the elements of the supplied array.
create	Creates an array whose elements are all initially the supplied value.
iter	Applies the supplied function to each element of an array.
length1	Returns the length of an array in the first dimension.
length2	Returns the length of an array in the second dimension.
map	Creates an array whose elements are the results of applying the supplied function to each of the elements of a supplied array.
mapi	
zeroCreate	Creates an array whose elements are all initially zero.

Table 8.2: Some built-in procedures in the Array2D module for arrays (from <https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/fsharp-core-library-reference>)

Notice that the indexing uses a slightly different notation '[,]' and the length functions are also slightly different. The statement `A.Length` would return the total number of elements in the array, in this case 12.

¹

There are a bit few built-in procedures for 2 dimensional array types, some of which are summarized in Table 8.2

8.4 Sequences

¹note that `A.[1,*]` is a `Array` but `A.[1..1,*]` is an `Array2D`.

Part II

Imperative programming

Chapter 9

Exceptions

9.1 Exception Handling

Exception handling allows programmers to catch and handle errors whenever an application enters an invalid state.

...

Chapter 10

Testing programs

...

Chapter 11

Input/Output

Reading and writing to files and the console window.

11.1 Console I/O

...

11.2 File I/O

```
let getAFileName () =
    let mutable filename = Unchecked.defaultof<string>
    let mutable fileExists = false
    while not(fileExists) do
        System.Console.Write("Enter Filename: ")
        filename <- System.Console.ReadLine()
        fileExists <- System.IO.File.Exists filename
    filename

let listOfFiles = System.IO.Directory.GetFiles(".")
printfn "Directory contains: %A" listOfFiles
let filename = getAFileName ()
printfn "You typed: %s" filename
```

```
let rec printFile (reader : System.IO.StreamReader) =
    if not(reader.EndOfStream) then
        let line = reader.ReadLine ()
        printfn "%s" line
        printFile reader

let filename = "readFile.fsx"
let reader = System.IO.File.OpenText filename
printFile reader

let rec printFile (reader : System.IO.StreamReader) =
    if not(reader.EndOfStream) then
        let line = reader.ReadLine ()
        printfn "%s" line
```

```

    printFile reader

let filename = "readFile.fsx"
let reader = System.IO.File.OpenText filename
printFile reader

```

Listing 11.1: readFile.fsx -

```

let rec readFile (stream : System.IO.StreamReader) =
    if not(stream.EndOfStream) then
        (stream.ReadLine ()) :: (readFile stream)
    else
        []

let rec writeFile (stream : System.IO.StreamWriter) text =
    match text with
    | (l : string) :: ls ->
        stream.WriteLine l
        writeFile stream ls
    | _ -> ()

let reverseString (s : string) =
    System.String(Array.rev (s.ToCharArray()))

let inputStream = System.IO.File.OpenText "reverseFile.fsx"
let text = readFile inputStream
let reverseText = List.map reverseString (List.rev text)
let outputStream = System.IO.File.CreateText "xsf.eliFesrever"
writeFile outputStream reverseText
outputStream.Close ()
printfn "%A" reverseText

```

```

["txeTesrever "A%" nftnirp"; ")( esolC.maertStuptuo";
"txeTesrever maertStuptuo eliFetirw";
""reverseFile.fsx" txeTetaerC.eliF.OI.metsyS = maertStuptuo tel";
")txet ver.tsiL( gnirtSesrever pam.tsiL = txeTesrever tel";
"maertStupni eliFdaer = txet tel";
""xsf.eliFesrever" txeTnepO.eliF.OI.metsyS = maertStupni tel"; "";
"))(yarrArahCoT.s( ver.yarrA(gnirtS.metsyS ";
"= )gnirts : s( gnirtSesrever tel"; ""; ")( >- _ | ";
"sl maerts eliFetirw "; "l eniLetirW.maerts ";
">- sl :: )gnirts : l( | "; "htiw txet hctam ";
"= txet )retirWmaertS.OI.metsyS : maerts( eliFetirw cer tel"; "";
"][ ";
"esle "; "maerts eliFdaer( :: ))( eniLdaeR.maerts( ";
"neht )maertSfOdnE.maerts(ton fi ";
"= )redaeRmaertS.OI.metsyS : maerts( eliFdaer cer tel"]

```

Listing 11.2: reverseFile.fsx -

Chapter 12

Graphical User Interfaces

...

Chapter 13

The Collection

13.1 Mutable Collections

`System.Collections.Generic`

13.1.1 Mutable lists

`List`, `LinkedList`

13.1.2 Stacks

`Stack`

13.1.3 Queues

`Queue`

13.1.4 Sets and dictionaries

`HashSet`, and `Dictionary` from

Chapter 14

Imperative programming

14.1 Introduction

Imperativ programming focusses on how a problem is to be solved as a list of *statements* and a set of *states*, where states may change over time. An example is a baking recipe, e.g.,

1. Mix yeast with water
2. Stir in salt, oil, and flour
3. Knead until the dough has a smooth surface
4. Let the dough rise until it has double size
5. Shape dough into a loaf
6. Let the loaf rise until double size
7. Bake in oven until the bread is golden brown

Each line in this example consists of one or more statements that are to be executed, and while executing them states such as size of the dough, color of the bread changes, and some execution will halt execution until certain conditions of these states are fulfilled, e.g., the bread will not be put into the oven for baking before it has risen sufficiently.

Statements may be grouped into procedures, and structuring imperative programs heavily into procedures is called *Procedural programming*, which is sometimes considered as a separate paradigm from imperative programming. *Object oriented programming* is an extension of imperative programming, where statements and states are grouped into classes and will be treated elsewhere.

Almost all computer hardware is designed for *machine code*, which is a common term used for many low-level computer programming language, and almost all machine languages follow the imperative programming paradigm.

Functional programming may be considered a subset of imperative programming, in the sense that functional programming does not include the concept of a state, or one may think of functional programming as only have one unchanging state. Functional programming has also a bigger focus on what should be solved, by declaring rules but not explicitly listing statements describing how these rules should be combined and executed in order to solve a given problem. Functional programming will be treated elsewhere.

14.2 Generating random texts

14.2.1 0'th order statistics

- Imperativ programming
- statements
- states

- Procedural programming
- Object oriented programming
- machine code
- Functional programming

```

let histToCumulativeProbability hist =
    let appendSum (acc : float array) (elem : int) =
        let sum =
            if acc.Length = 0 then
                float elem
            else
                acc.[acc.Length-1] + (float elem)
        Array.append acc [| sum |]

    let normalizeProbability k v = v/k

    let cumSum = Array.fold appendSum Array.empty<float> hist
    if cumSum.[cumSum.Length - 1] > 0.0 then
        Array.map (normalizeProbability cumSum.[cumSum.Length - 1])
            cumSum
    else
        Array.create cumSum.Length (1.0 / (float cumSum.Length))

let lookup (hist : float array) (v : float) =
    Array.findIndex (fun h -> h > v) hist

let countEqual A v =
    Array.fold (fun acc elem -> if elem = v then acc+1 else acc) 0 A

let intToIdx i = i - (int ' ')
let idxToInt i = i + (int ' ')

let legalIndex size idx =
    (0 <= idx) && (idx <= size - 1)

let analyzeFile (reader : System.IO.StreamReader) size pushForward
    =
    let hist = Array.create size 0
    let mutable c = Unchecked.defaultof<int>
    while not(reader.EndOfStream) do
        c <- pushForward (reader.Read ())
        if legalIndex size c then
            hist.[c] <- hist.[c] + 1
    hist

let sampleFromCumulativeProbability cumulative noSamples =
    let rnd = System.Random ()
    let rndArray = Array.init noSamples (fun _ -> rnd.NextDouble ())
    Array.map (lookup cumulative) rndArray

let filename = "randomTextOrder0.fsx"
let noSamples = 200
let histSize = 126 - 32 + 1

let reader = System.IO.File.OpenText filename
let hist = analyzeFile reader histSize intToIdx

```

```

reader.Close ()
let idxValue = Array.mapi (fun i v -> (idxToInt i, v)) hist
printfn "%A" idxValue
printfn "%d zeros out of %d elements" (countEqual hist 0) hist.
    Length
let cumulative = histToCumulativeProbability hist
let rndIdx = sampleFromCumulativeProbability cumulative noSamples
let rndInt = Array.map idxToInt rndIdx
let rndChar = Array.map (fun v -> char v) rndInt
let text = System.String.Concat rndChar // System.String is not the
    same as String !!!
printfn "%A" text

```

```

[(32, 339); (33, 3); (34, 8); (35, 0); (36, 0); (37, 4); (38, 2);
(39, 4);
(40, 27); (41, 27); (42, 0); (43, 5); (44, 1); (45, 13); (46, 42)
; (47, 4);
(48, 11); (49, 9); (50, 3); (51, 1); (52, 0); (53, 0); (54, 1);
(55, 0);
(56, 0); (57, 0); (58, 5); (59, 0); (60, 6); (61, 31); (62, 9);
(63, 0);
(64, 0); (65, 19); (66, 0); (67, 8); (68, 1); (69, 3); (70, 7);
(71, 0);
(72, 0); (73, 14); (74, 0); (75, 0); (76, 7); (77, 0); (78, 1);
(79, 5);
(80, 6); (81, 0); (82, 3); (83, 26); (84, 9); (85, 1); (86, 2);
(87, 0);
(88, 0); (89, 0); (90, 0); (91, 6); (92, 0); (93, 6); (94, 0);
(95, 1);
(96, 0); (97, 100); (98, 14); (99, 41); (100, 50); (101, 143);
(102, 29);
(103, 12); (104, 38); (105, 87); (106, 0); (107, 5); (108, 89);
(109, 63);
(110, 75); (111, 50); (112, 26); (113, 2); (114, 88); (115, 47);
(116, 121);
(117, 55); (118, 18); (119, 3); (120, 21); (121, 31); (122, 12);
(123, 0);
(124, 2); (125, 0); (126, 0)]
29 zeros out of 95 elements
"i)iraru n P.tnSm llcl =nCvte i/ ctraPe2 eF d.l m rt=e gusebtcu
t0r uaaealuLtreltqrzrSrtP.de s a gxs (oo=elgol Rse
iuFhet0meteeetut0frti .er a iI P ccmyeuoe<sp lnrtp lehmroIe-
oS(eee(1 s panr m,s"

```

Listing 14.1: randomTextOrder0.fsx -

14.2.2 1'th order statistics

```

let histToCumulativeProbability hist =
    let appendSum (acc : float array) (elem : int) =
        let sum =
            if acc.Length = 0 then
                float elem

```

```

        else
            acc.[acc.Length-1] + (float elem)
        Array.append acc [| sum |]

let normalizeProbability k v = v/k

let cumSum = Array.fold appendSum Array.empty<float> hist
if cumSum.[cumSum.Length - 1] > 0.0 then
    Array.map (normalizeProbability cumSum.[cumSum.Length - 1])
        cumSum
else
    Array.create cumSum.Length (1.0 / (float cumSum.Length))

let lookup (hist : float array) (v : float) =
    Array.findIndex (fun h -> h > v) hist

let countEqual A v =
    Array.fold (fun acc elem -> if elem = v then acc+1 else acc) 0 A

let intToIdx i = i - (int ' ')

let idxToInt i = i + (int ' ')

let legalIndex size idx =
    (0 <= idx) && (idx <= size - 1)

let analyzeFile (reader : System.IO.StreamReader) size pushForward
    =
    let hist = Array2D.create size size 0
    let mutable c1 = Unchecked.defaultof<int>
    let mutable c2 = Unchecked.defaultof<int>
    let mutable nRead = 0
    while not(reader.EndOfStream) do
        c2 <- pushForward (reader.Read ())
        if legalIndex size c2 then
            nRead <- nRead + 1
            if nRead >= 2 then
                hist.[c1,c2] <- hist.[c1,c2] + 1
            c1 <- c2;
    hist

let Array2DToArray (arr : 'T [,]) = arr |> Seq.cast<'T> |> Seq.
    toArray

let Array2DOfArray (a : 'T []) = Array2D.init 1 a.Length (fun i j
    -> a.[j])

let hist2DCumulativeProbability hist =
    let rows = Array2D.length1 hist
    let cols = Array2D.length2 hist
    let cumulative = Array2D.zeroCreate<float> rows cols
    for i = 0 to rows - 1 do
        let histi = Array2DOfArray (histToCumulativeProbability hist.[i

```

```

        ,*])
        Array2D.blit histi 0 0 cumulative i 0 1 cols
        cumulative

let marginal (hist : int [,]) =
    let rows = Array2D.length1 hist
    let sum = Array.zeroCreate rows
    for i = 0 to rows - 1 do
        sum.[i] <- Array.sum hist.[i,*]
    sum

let sampleFromCumulativeProbability (cumulative : float [,]) (
    margCumulative : float array) noSamples =
    let rnd = System.Random ()
    let samples = Array.zeroCreate<int> noSamples
    let mutable v = rnd.NextDouble ()
    let mutable i = Unchecked.defaultof<int>
    samples.[0] <- lookup margCumulative v
    for n = 1 to noSamples - 1 do
        v <- rnd.NextDouble ()
        i <- samples.[n - 1]
        samples.[n] <- lookup cumulative.[n, *] v
    samples

let filename = "randomTextOrder0.fsx"
let noSamples = 200
let histSize = 126 - 32 + 1

let reader = System.IO.File.OpenText filename
let hist = analyzeFile reader histSize intToIdx
reader.Close ()
let idxValue = Array2D.mapi (fun i j v -> (idxToInt i, idxToInt j,
    v)) hist
printfn "%A" (Array2DToArray idxValue)
printfn "%d zeros out of %d elements" (countEqual (Array2DToArray
    hist) 0) hist.Length
let margHist = marginal hist;
let margCumulative = histToCumulativeProbability margHist
let cumulative = hist2DCumulativeProbability hist
(*
let rndIdx = sampleFromCumulativeProbability cumulative
    margCumulative noSamples
let rndInt = Array.map idxToInt rndIdx
let rndChar = Array.map (fun v -> char v) rndInt
let text = System.String.Concat rndChar // System.String is not the
    same as String !!!
printfn "%A" text
*)

[(32, 32, 63); (32, 33, 1); (32, 34, 4); (32, 35, 0); (32, 36, 0);
    (32, 37, 1);
    (32, 38, 1); (32, 39, 4); (32, 40, 26); (32, 41, 0); (32, 42, 0);
    (32, 43, 4);

```

```

(32, 44, 0); (32, 45, 10); (32, 46, 0); (32, 47, 2); (32, 48, 5);
(32, 49, 6);
(32, 50, 1); (32, 51, 1); (32, 52, 0); (32, 53, 0); (32, 54, 0);
(32, 55, 0);
(32, 56, 0); (32, 57, 0); (32, 58, 5); (32, 59, 0); (32, 60, 4);
(32, 61, 29);
(32, 62, 2); (32, 63, 0); (32, 64, 0); (32, 65, 15); (32, 66, 0);
(32, 67, 0);
(32, 68, 0); (32, 69, 0); (32, 70, 0); (32, 71, 0); (32, 72, 0);
(32, 73, 0);
(32, 74, 0); (32, 75, 0); (32, 76, 0); (32, 77, 0); (32, 78, 0);
(32, 79, 0);
(32, 80, 0); (32, 81, 0); (32, 82, 0); (32, 83, 6); (32, 84, 0);
(32, 85, 1);
(32, 86, 0); (32, 87, 0); (32, 88, 0); (32, 89, 0); (32, 90, 0);
(32, 91, 1);
(32, 92, 0); (32, 93, 0); (32, 94, 0); (32, 95, 1); (32, 96, 0);
(32, 97, 13);
(32, 98, 0); (32, 99, 15); (32, 100, 1); (32, 101, 8); (32, 102,
6);
(32, 103, 0); (32, 104, 18); (32, 105, 20); (32, 106, 0); (32,
107, 1);
(32, 108, 11); (32, 109, 1); (32, 110, 7); (32, 111, 2); (32,
112, 2);
(32, 113, 0); (32, 114, 12); (32, 115, 10); (32, 116, 7); (32,
117, 0);
(32, 118, 9); (32, 119, 1); (32, 120, 0); (32, 121, 0); (32, 122,
1);
(32, 123, 0); (32, 124, 1); (32, 125, 0); (32, 126, 0); (33, 32,
0);
(33, 33, 2); (33, 34, 0); (33, 35, 0); (33, 36, 0); ...|]
8640 zeros out of 9025 elements

```

Listing 14.2: randomTextOrder1.fsx -

Part III

Declarative programming

Chapter 15

Functional programming

Part IV

Structured programming

Chapter 16

Object-oriented programming

Part V

Appendix

Appendix A

Number systems on the computer

A.1 Binary numbers

Humans like to use the *decimal number* system for representing numbers. Decimal numbers are *base 10* means that for a number consisting of a sequence of digits separated by a *decimal point*, where each *digit* can have values $d \in \{0, 1, 2, \dots, 9\}$ and the weight of each digit is proportional to its place in the sequence of digits w.r.t. the decimal point, i.e., the number $357.6 = 3 \cdot 10^2 + 5 \cdot 10^1 + 7 \cdot 10^0 + 6 \cdot 10^{-1}$ or in general:

· decimal number
· base
· decimal point
· digit

$$v = \sum_{i=-m}^n d_i 10^i \quad (\text{A.1})$$

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

$$v = \sum_{i=-m}^n b_i 2^i \quad (\text{A.2})$$

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. Due to typical organization of computer memory, 8 binary digits is called a *byte*, and 32 digits a *word*.

· byte
· word
· octal
· hexadecimal

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

A.2 IEEE 754 floating point standard

The set of real numbers also called *reals* includes all fractions and irrational numbers. It is infinite in size both in the sense that there is no largest nor smallest number and between any 2 given numbers there are infinitely many numbers. Reals are widely used for calculation, but since any computer only has finite memory, it is impossible to represent all possible reals. 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,

· reals

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

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

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 A.1: A list of the intergers 0–63 in decimal, binary, octal, and hexadecimal.

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 ,

$$e = \sum_{i=1}^{11} e_i 2^{11-i}, \quad (\text{A.3})$$

$$m = \sum_{j=1}^{52} m_j 2^{-j}. \quad (\text{A.4})$$

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,

$$v = (-1)^s (1 + m) 2^{e-1023} \quad (\text{A.5})$$

with the exception that

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

· subnormals
· NaN
· not a number

where $e = 2^{11} - 1 = 11111111111_2 = 2047$. The largest and smallest number that is not infinity is thus

$$e = 2^{11} - 2 = 2046 \quad (\text{A.6})$$

$$m = \sum_{j=1}^{52} 2^{-j} = 1 - 2^{-52} \simeq 1. \quad (\text{A.7})$$

$$v_{\max} = \pm (2 - 2^{-52}) 2^{1023} \simeq \pm 2^{1024} \simeq \pm 10^{308} \quad (\text{A.8})$$

The density of numbers varies in such a way that when $e - 1023 = 52$, then

$$v = (-1)^s \left(1 + \sum_{j=1}^{52} m_j 2^{-j} \right) 2^{52} \quad (\text{A.9})$$

$$= \pm \left(2^{52} + \sum_{j=1}^{52} m_j 2^{-j} 2^{52} \right) \quad (\text{A.10})$$

$$= \pm \left(2^{52} + \sum_{j=1}^{52} m_j 2^{52-j} \right) \quad (\text{A.11})$$

$$\stackrel{k=52-j}{=} \pm \left(2^{52} + \sum_{k=51}^0 m_{52-k} 2^k \right) \quad (\text{A.12})$$

which are all integers in the range $2^{52} \leq |v| < 2^{53}$. When $e - 1023 = 53$, then the same calculation gives

$$v \stackrel{k=53-j}{=} \pm \left(2^{53} + \sum_{k=52}^1 m_{53-k} 2^k \right) \quad (\text{A.13})$$

which are every second integer in the range $2^{53} \leq |v| < 2^{54}$, and so on for larger e . When $e - 1023 = 51$, then the same calculation gives,

$$v \stackrel{k=51-j}{=} \pm \left(2^{51} + \sum_{k=50}^{-1} m_{51-k} 2^k \right) \quad (\text{A.14})$$

which gives a distance between numbers of $1/2$ in the range $2^{51} \leq |v| < 2^{52}$, and so on for smaller e . Thus we may conclude that the distance between numbers in the interval $2^n \leq |v| < 2^{n+1}$ is 2^{n-52} , for $-1022 = 1 - 1023 \leq n < 2046 - 1023 = 1023$. For subnormals the distance between numbers are

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

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

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

$$\stackrel{k=-j-1022}{=} \pm \left(\sum_{j=-1023}^{-1074} m_{-k-1022} 2^k \right) \quad (\text{A.18})$$

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

Appendix B

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 others. These symbols are in short called characters, and represents a mapping between numbers, also known as codes, and a pictorial representation of the character. E.g., the ASCII code for the letter 'A' is 65. These mappings are for short called character sets, and due to differences in natural languages and symbols used across the globe, many different character sets are in use. E.g., the English alphabet contains the letters 'a' to 'z', which is shared by many other European languages, but which have other symbols and accents for example, Danish has further the letters 'æ', 'ø', and 'å'. Many non-european languages have completely different symbols, where Chinese character set is probably the most extreme, where some definitions contains 106,230 different characters albeit only 2,600 are included in the official Chinese language test at highest level.

Presently, the most common character set used is Unicode Transformation Format (UTF), whose most popular encoding schemes are 8-bit (UTF-8) and 16-bit (UTF-16). Many other character sets exists, and many of the later builds 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.

B.1 ASCII

The *American Standard Code for Information Interchange* (ASCII) [4], is a 7 bit code tuned for the letters of the english language, numbers, punctuation symbols, control codes and space, see Tables B.1 and B.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 characters is not universally agreed upon.

The code order is known as *ASCIIbetical order*, and it is sometimes used to perform arithmetic on codes, e.g., an upper case letter with code c may be converted to lower case by adding 32 to its code. The ASCIIbetical order also has consequence for sorting, i.e., when sorting characters according to their ASCII code, then 'A' comes before 'a', which comes before the symbol '{'.

- American Standard Code for Information Interchange
- ASCII
- ASCIIbetical order

B.2 ISO/IEC 8859

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

- Latin1

x0+0x	00	10	20	30	40	50	60	70
00	NUL	DLE	SP	0	@	P	'	p
01	SOH	DC1	!	1	A	Q	a	q
02	STX	DC2	"	2	B	R	b	r
03	ETX	DC3	#	3	C	S	c	s
04	EOT	DC4	\$	4	D	T	d	t
05	ENQ	NAK	%	5	E	U	e	u
06	ACK	SYN	&	6	F	V	f	v
07	BEL	ETB	,	7	G	W	g	w
08	BS	CAN	(8	H	X	h	x
09	HT	EM)	9	I	Y	i	y
0A	LF	SUB	*	:	J	Z	j	z
0B	VT	ESC	+	;	K	[k	{
0C	FF	FS	,	<	L	\	l	
0D	CR	GS	-	=	M]	m	}
0E	SO	RS	.	>	N	^	n	~
0F	SI	US	/	?	O	_	o	DEL

Table B.1: ASCII

B.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 requires 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 it are the first 128 code points is called the *Basic Latin block* and are identical to ASCII, see Table B.1, and code points 128-255 is called the *Latin-1 Supplement block*, and are identical to the upper range of ISO 8859-1, see Table B.3. Presently more than 128,000 code points covering 135 modern and historic writing systems.

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 is “U+0041”, and in this text it is visualized as ‘A’. More digits are used for code points of the remaining planes.

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 compiles, interpreters and operating systems.

- Unicode Standard
- code point
- blocks
- Basic Multilingual plane
- Basic Latin block
- Latin-1 Supplement block

- UTF-8

- UTF-16

Code	Description
NUL	Null
SOH	Start of heading
STX	Start of text
ETX	End of text
EOT	End of transmission
ENQ	Enquiry
ACK	Acknowledge
BEL	Bell
BS	Backspace
HT	Horizontal tabulation
LF	Line feed
VT	Vertical tabulation
FF	Form feed
CR	Carriage return
SO	Shift out
SI	Shift in
DLE	Data link escape
DC1	Device control one
DC2	Device control two
DC3	Device control three
DC4	Device control four
NAK	Negative acknowledge
SYN	Synchronous idle
ETB	End of transmission block
CAN	Cancel
EM	End of medium
SUB	Substitute
ESC	Escape
FS	File separator
GS	Group separator
RS	Record separator
US	Unit separator
DEL	Delete

Table B.2: ASCII symbols.

x0+0x	80	90	A0	B0	C0	D0	E0	F0
00			NBSP	°	À	Ð	à	ð
01			¡	±	Á	Ñ	á	ñ
02			¢	²	Â	Ò	â	ò
03			£	³	Ã	Ó	ã	ó
04			¤	´	Ä	Ô	ä	ô
05			¥	µ	Å	Õ	å	õ
06			¦	¶	Æ	Ö	æ	ö
07			§	·	Ç	×	ç	÷
08			¨	¸	È	Ø	è	ø
09			©	¹	É	Ù	é	ù
0a			ª	º	Ê	Ú	ê	ú
0b			«	»	Ë	Û	ë	û
0c			¬	$\frac{1}{4}$	Ì	Ü	ì	ü
0d			SHY	$\frac{1}{2}$	Í	Ý	í	ý
0e			®	$\frac{3}{4}$	Î	Þ	î	þ
0f			¯	¸	Ï	ß	ï	ÿ

Table B.3: ISO-8859-1 (latin1) non-ASCII part. Note that the codes 7f – 9f are undefined.

Code	Description
NBSP	Non-breakable space
SHY	Soft hyphen

Table B.4: ISO-8859-1 special symbols.

Appendix C

A brief introduction to Extended Backus-Naur Form

Extended Backus-Naur Form (EBNF) is a language to specify programming languages in. The name is a tribute to John Backus who used it to describe the syntax of ALGOL58 and Peter Naur for his work on ALGOL 60.

An EBNF consists of *terminal symbols* and *production rules*. Examples of typical terminal symbol are characters, numbers, punctuation marks, and whitespaces, e.g.,

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

A production rule specifies a method of combining other production rules and terminal symbols, e.g.,

```
number = { digit } ;
```

A proposed standard for EBNF (proposal ISO/IEC 14977, <http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf>) is,

'=' definition, e.g.,

```
zero = "0" ;
```

here **zero** is the terminal symbol 0.

',' concatenation, e.g.,

```
one = "1" ;  
eleven = one , one ;
```

here **eleven** is the terminal symbol 11.

',' termination of line

'|' alternative options, e.g.,

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "  
9" ;
```

here **digit** is the single character terminal symbol, such as 3.

'[...]' optional, e.g.,

- Extended Backus-Naur Form
- EBNF
- terminal symbols
- production rules

```

zero = "0" ;
nonZeroDigit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
               "9" ;
nonZero = [ zero ], nonZeroDigit

```

here `nonZero` is a non-zero digit possibly preceded by zero, such as 02.

'{ ... }' repetition zero or more times, e.g.,

```

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "
9" ;
number = digit, { digit }

```

here `number` is a word consisting of 1 or more digits, such as 12.

'(...)' grouping, e.g.,

```

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "
9" ;
number = digit, { digit }
expression = number, { ( "+" | "-" ), number };

```

here `expression` is a number or a sum of numbers such as 3 + 5.

'" ... "' a terminal string, e.g.,

```

string = "abc" ;

```

'" ... "' a terminal string, e.g.,

```

string = 'abc' ;

```

'(* ... *)' a comment (* ... *)

```

(* a binary digit *) digit = "0" | "1" (* from this all numbers
may be constructed *) ;

```

Everything inside the comments are not part of the formal definition.

'? ... ?' special sequence, a notation reserved for future extensions of EBNF.

'-' exception, e.g.,

```

letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H"
        | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q"
        | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" ;
vowel = "A" | "E" | "I" | "O" | "U" ;
consonant = letter - vowel ;

```

here `consonant` are all letters except vowels.

The proposal allows for identifiers that includes space, but often a reduced form is used, where identifiers are single words, in which case the concatenation symbol `,` is omitted. Likewise, the termination symbol `;` is often replaced with the new-line character, and if long lines must be broken, then indentation is used to signify continuation.

In this relaxed EBNF, the EBNF syntax itself can be expressed in EBNF as,

```

letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H"
        | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q"
        | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z"
        | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h"
        | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q"
        | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
symbol = "[" | "]" | "{" | "}" | "(" | ")" | "<" | ">"
        | "'" | '"' | "=" | "|" | "." | "," | ";"
underscore = "_"
identifier = letter { letter | digit | underscore }
character = letter | digit | symbol | underscore
string = character { character }
terminal = '"' string '"' | "'" string "'"
rhs = identifier
      | terminal
      | "[" rhs "]"
      | "{" rhs "}"
      | "(" rhs ")"
      | rhs "|" rhs
(* | rhs "," rhs *)
rule = identifier "=" rhs (* ";" *)
grammar = rule { rule }

```

Here the comments demonstrate, the relaxed modification.

Bibliography

- [1] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.
- [2] Programming Research Group. Specifications for the ibm mathematical formula translating system, fortran. Technical report, Applied Science Division, International Business Machines Corporation, 1954.
- [3] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.
- [4] 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/>.
- [5] George Pólya. *How to solve it*. Princeton University Press, 1945.

Index

- it, 19
- American Standard Code for Information Inter-
change, 63
- and, 22
- ASCII, 63
- ASCIIbetical order, 63
- base, 59
- Basic Latin block, 64
- Basic Multilingual plane, 64
- binary, 59
- binary64, 59
- binding, 9
- bit, 59
- blocks, 64
- byte, 59
- class, 16
- code point, 64
- compiled, 7
- console, 7
- currying, 29
- debugging, 8
- decimal number, 59
- decimal point, 59
- Declarative programming, 4
- digit, 59
- double, 59
- downcasting, 16
- EBNF, 67
- encapsulation, 25
- exception, 19
- executable file, 7
- expression, 9
- expressions, 5
- Extended Backus-Naur Form, 67
- format string, 9
- function, 11
- Functional programming, 5, 48
- functions, 5
- hexadecimal, 59
- IEEE 754 double precision floating-point format,
59
- Imperativ programming, 48
- Imperative programming, 4
- implementation file, 7
- infix notation, 16
- interactive, 7
- jagged arrays, 38
- keyword, 9
- Latin-1 Supplement block, 64
- Latin1, 63
- literal, 13
- literal type, 13
- literals, 13
- machine code, 48
- member, 15
- modules, 7
- Mutable data, 23
- namespace, 16
- NaN, 61
- nested scope, 11
- not, 22
- not a number, 61
- Object oriented programming, 48
- Object-oriented programming, 5
- objects, 5
- octal, 59
- operand, 27
- operator, 16, 27
- or, 22
- overflow, 19
- overshadow, 11
- precedence, 16
- primitive types, 13
- Procedural programming, 48
- procedure, 29
- production rules, 67
- reals, 59
- reference cells, 25

- rounding, 16
- run-time error, 20

- scope, 11, 17
- script file, 7
- script-fragments, 7
- side-effects, 26
- signature file, 7
- slicing, 36
- state, 4
- statement, 9
- statements, 4, 48
- states, 48
- string, 9
- Structured programming, 5
- subnormals, 61

- terminal symbols, 67
- token, 11
- truth table, 22
- type, 9, 13
- type casting, 15
- type declaration, 9
- type inference, 8, 9

- underflow, 19
- Unicode Standard, 64
- unit-testing, 8
- upcasting, 16
- UTF-16, 64
- UTF-8, 64

- variable, 23

- word, 59