

# Learning to program with F#

Jon Sparring

July 31, 2016

# Contents

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Preface</b>                              | <b>4</b>  |
| <b>2</b> | <b>Introduction</b>                         | <b>5</b>  |
| <b>I</b> | <b>F# basics</b>                            | <b>7</b>  |
| <b>3</b> | <b>Executing F# code</b>                    | <b>8</b>  |
| 3.1      | Source code . . . . .                       | 8         |
| 3.2      | Executing programs . . . . .                | 8         |
| <b>4</b> | <b>Quick-start guide</b>                    | <b>10</b> |
| <b>5</b> | <b>Using F# as a calculator</b>             | <b>14</b> |
| 5.1      | Literals and basic types . . . . .          | 14        |
| 5.2      | Operators on basic types . . . . .          | 19        |
| 5.3      | Boolean arithmetic . . . . .                | 24        |
| 5.4      | Integer arithmetic . . . . .                | 25        |
| 5.5      | Floating point arithmetic . . . . .         | 26        |
| 5.6      | Char and string arithmetic . . . . .        | 27        |
| <b>6</b> | <b>Constants, functions, and variables</b>  | <b>30</b> |
| 6.1      | Values . . . . .                            | 32        |
| 6.2      | Non-recursive functions . . . . .           | 35        |
| 6.3      | User-defined operators . . . . .            | 38        |
| 6.4      | Printf . . . . .                            | 40        |
| 6.5      | Variables . . . . .                         | 42        |
| 6.6      | In-code documentation . . . . .             | 45        |
| <b>7</b> | <b>Controlling program flow</b>             | <b>50</b> |
| 7.1      | For and while loops . . . . .               | 50        |
| 7.2      | Conditional expressions . . . . .           | 53        |
| 7.2.1    | Programming intermezzo . . . . .            | 54        |
| 7.3      | Pattern matching . . . . .                  | 55        |
| 7.4      | Recursive functions . . . . .               | 57        |
| <b>8</b> | <b>Tuples, Lists, Arrays, and Sequences</b> | <b>59</b> |
| 8.1      | Tuples . . . . .                            | 60        |
| 8.2      | Lists . . . . .                             | 62        |
| 8.3      | Arrays . . . . .                            | 65        |
| 8.3.1    | 1 dimensional arrays . . . . .              | 65        |
| 8.3.2    | Multidimensional Arrays . . . . .           | 68        |
| 8.4      | Sequences . . . . .                         | 70        |

|            |  |            |
|------------|--|------------|
| <b>II</b>  | <b>Imperative programming</b>                            | <b>72</b>  |
| <b>9</b>   | <b>Exceptions</b>  | <b>73</b>  |
| 9.1        | Exception Handling . . . . .                             | 73         |
| <b>10</b>  | <b>Testing programs</b>                                  | <b>74</b>  |
| <b>11</b>  | <b>Input/Output</b>                                      | <b>75</b>  |
| 11.1       | Console I/O . . . . .                                    | 75         |
| 11.2       | File I/O . . . . .                                       | 75         |
| <b>12</b>  | <b>Graphical User Interfaces</b>                         | <b>77</b>  |
| <b>13</b>  | <b>The Collection</b>                                    | <b>78</b>  |
| 13.1       | System.String . . . . .                                  | 78         |
| 13.2       | List, arrays, and sequences . . . . .                    | 83         |
| 13.3       | Mutable Collections . . . . .                            | 85         |
| 13.3.1     | Mutable lists . . . . .                                  | 85         |
| 13.3.2     | Stacks . . . . .   | 85         |
| 13.3.3     | Queues . . . . .   | 85         |
| 13.3.4     | Sets and dictionaries . . . . .                          | 86         |
| <b>14</b>  | <b>Imperative programming</b>                            | <b>87</b>  |
| 14.1       | Introduction . . . . .                                   | 87         |
| 14.2       | Generating random texts . . . . .                        | 87         |
| 14.2.1     | 0'th order statistics . . . . .                          | 87         |
| 14.2.2     | 1'th order statistics . . . . .                          | 89         |
| <b>III</b> | <b>Declarative programming</b>                           | <b>92</b>  |
| <b>15</b>  | <b>Types and measures</b>                                | <b>93</b>  |
| 15.1       | Unit of Measure . . . . .                                | 93         |
| <b>16</b>  | <b>Functional programming</b>                            | <b>96</b>  |
| <b>IV</b>  | <b>Structured programming</b>                            | <b>97</b>  |
| <b>17</b>  | <b>Namespaces and Modules</b>                            | <b>98</b>  |
| <b>18</b>  | <b>Object-oriented programming</b>                       | <b>100</b> |
| <b>V</b>   | <b>Appendix</b>  | <b>101</b> |
| <b>A</b>   | <b>Number systems on the computer</b>                    | <b>102</b> |
| A.1        | Binary numbers . . . . .                                 | 102        |
| A.2        | IEEE 754 floating point standard . . . . .               | 102        |
| <b>B</b>   | <b>Commonly used character sets</b>                      | <b>106</b> |
| B.1        | ASCII . . . . .  | 106        |
| B.2        | ISO/IEC 8859 . . . . .                                   | 106        |
| B.3        | Unicode . . . . .  | 107        |
| <b>C</b>   | <b>A brief introduction to Extended Backus-Naur Form</b> | <b>110</b> |

|                           |            |
|---------------------------|------------|
| <b>D Language Details</b> | <b>113</b> |
| <b>Bibliography</b>       | <b>115</b> |
| <b>Index</b>              | <b>116</b> |

# Chapter 1

## Preface

...<sup>1</sup>

---

<sup>1</sup>Something 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. <sup>1</sup>

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

---

<sup>1</sup>Should 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*

**.fsscript** 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 `";;"` lexeme, 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 `";;"` lexeme:

```

$ 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`,  $1.88s \gg 0.05s$ .

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

## 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 newline (LF possibly preceded by CR, 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:

```
> let a = 357;;
```

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

· format string  
· string

· type

· type declaration  
· type inference

```

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, Inferred types are given as part of the response from the interpreter.

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/repositories/fsharpNotes/src/stdin(4,13): error FS0001: The
    type 'float' does not match the type 'int'

```

**Listing 4.4:** fsharpi, Mixing types is often not allowed.

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 have a *lexical scope*. A scope is an area in a program, where a binding is valid, and lexical scope means that when a binding is used, then its value is substituted at the place of binding regardless of whether its value is rebound later in the text. Further, at the outer most level, rebinding is not allowed. If attempted, then F# will return an error as, e.g.,<sup>1</sup>

· lexical scope

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

```
/Users/sporring/repositories/fsharpNotes/src/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, Names may be reused when separated by the lexeme ; ;.

then apparently rebinding is valid. The difference is that the ; ; *lexeme* defines a new nested *scope*.<sup>2</sup> A lexeme is a letter or a word, which the F# considers as an atomic unit. 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.,

· ; ;  
· lexeme  
· scope  
· nested scope  
· overshadow

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

  let a = 864;;
  ----^

/Users/sporring/repositories/fsharpNotes/src/stdin(3,5): error FS0037:
Duplicate definition of value 'a'
```

**Listing 4.7:** fsharpi, Inside a block, names may not be reused.

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 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.

<sup>1</sup>When command is omitted, then error messages have unwanted blank lines.

<sup>2</sup>Language change: Spec 4.0 p. 15.1 talks about blocks instead of scopes.

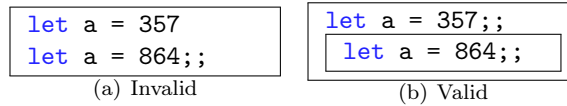


Figure 4.1: Binding of the the same name in the same scope is invalid in F# 2, but valid in a different scopes. In (a) the two bindings are in the same scope, which is invalid, while in (b) the bindings are in separate scopes by the extra `;;` lexeme, which is valid.

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

**Listing 4.9:** fsharpi, Types are inferred in blocks, and F# tends to prefer integers.

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, Defining a function together with its use, makes F# infer the appropriate types.

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

## Using F# as a calculator

### 5.1 Literals and basic types

All programs rely on processing of data, and an essential property of data is its *type*. A *literal* is a fixed value such as "3", and if we type the number 3 in an interactive session at the input prompt, then F# responds as follows,

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

**Listing 5.1:** fsharp, Typing the number 3.

What this means is that F# has inferred the type to be *int* and bound it to the identifier *it*. Rumor has it, that the identifier *it* is an abbreviation for 'irrelevant'. For more on binding and identifiers see Chapter 6. Types matter, since the operations that can be performed on integers are quite different from those that can be performed on, e.g., strings. I.e.,

```
> 3;;  
val it : int = 3  
> 3.0;;  
val it : float = 3.0  
> '3';;  
val it : char = '3'  
> "3";;  
val it : string = "3"
```

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

Each literal represent the number 3, but their types are different, and hence they are quite different values. The types *int* for integer numbers, *float* for floating point numbers, *char* for characters, and *string* for strings of characters are the most common types of literals. A table of all *basic types* predefined in F# is given in Table 5.1. Besides these built-in types, F# is designed such that it is easy to define new types.

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 value, which each digit represents is proportional to its position. The part before the decimal point is called the *whole part* and the part after is called the *fractional part* of the number. The whole part without a decimal point and a fractional part is called an *integer number*. As an example 35.7 is a decimal number, whose value is  $3 \cdot 10^1 + 5 \cdot 10^0 + 7 \cdot 10^{-1}$ . In F# a decimal number is called a *floating point number* and in this text we use *Extended Backus-Naur Form (EBNF)* to describe the grammar of F#, the decimal number just described is given as,

```
dDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"  
dInt = dDigit {dDigit}
```

· type  
· literal

· int  
· it

· float  
· char  
· string  
· basic types  
· decimal number  
· base  
· decimal point  
· digit  
· whole part  
· fractional part  
· integer number  
· floating point number  
· Extended Backus-Naur Form  
· EBNF

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

Table 5.1: List of basic types. The most commonly used types are highlighted in bold. For at description of integer see Appendix A.1, for floating point numbers see Appendix A.2, for ASCII and Unicode characters see Appendix B, for objects see Chapter 18, and for exceptions see Chapter 9.



```
dFloat = dInt "." {dDigit}
```

meaning that a dDigit is either "0" or "1" or ... or "9", an dInt is 1 or more dDigit, and a dFloat is 1 or more digits, a dot and 0 or more digits. There is no space between the digits and between digits and the dot. So 3, 049 are examples of integers, 34.89 3. are examples of floats, while .5 is neither. Floating point numbers may alternatively be given using *scientific notation*, such as 3.5e-4 and 4e2, which means the number  $3.5 \cdot 10^{-4} = 0.00035$  and  $4 \cdot 10^2 = 400$ . To describe this in EBNF we write

· scientific notation

```
sFloat = (dInt | dFloat) ("e" | "E") ["+" | "-"] dInt
float = dFloat | sFloat
```

Note that the number before the lexeme e may be an dInt or a dFloat, but the exponent value must be an dInt.

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. E.g., the binary number  $101.01_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 5.25$ . Binary numbers are closely related to *octal* and *hexadecimal numbers*, where octals uses 8 as basis and can be written in binary using 3 bits, while hexadecimal numbers uses 16 as basis and can be written in binary using 4 bits. Octals and hexadecimal numbers thus conveniently serve as shorthand for the much longer binary representation. F# has a syntax for writing integers on binary, octal, decimal, and hexadecimal numbers as,

· bit  
· binary number  
· octal number  
· hexadecimal number

```
bDigit = "0" | "1"
oDigit = bDigit | "2" | "3" | "4" | "5" | "6" | "7"
dDigit = oDigit | "8" | "9"
xDigit =
    dDigit
    | "A" | "B" | "C" | "D" | "E" | "F"
    | "a" | "b" | "c" | "d" | "e" | "f"
dInt = dDigit {dDigit}
bitInt = "0" ("b" | "B") bDigit {bDigit}
octInt = "0" ("o" | "O") oDigit {oDigit}
hexInt = "0" ("x" | "X") xDigit {xDigit}
xInt = bitInt | octInt | hexInt
int = dInt | xInt
```

For example 367 is an dInt, 0b101101111, 0o557, and 0x16f is a bitInt, octInt, and hexInt, i.e., a binary, an octal, and a hexadecimal number, they are examples of an xInt and representations of the same number 367. In contrast, 0b12 and ff are neither an dInt nor an xInt.

A *character* is a *Unicode code point*, and character literals are enclosed in single quotation marks, see Appendix B.3 for a description of code points.<sup>1</sup> The EBNF for characters is,

· character  
· Unicode  
· code point

```
escapeCodePoint =
    "\u" xDigit xDigit xDigit xDigit
    | "\U" xDigit xDigit xDigit xDigit xDigit xDigit xDigit xDigit
    | "\" dDigit dDigit dDigit
escapeChar =
    "\"" ("b" | "n" | "r" | "t" | "\" | "'" | '"' | "a" | "f" | "v")
    | escapeCodePoint
char = "'" codePoint | escapeChar "'"
```

where codePoint is a UTF8 encoding of a char. The escape characters escapeChar are special sequences that are interpreted as a single code point shown in Table 5.2. The trigraph \DDD uses decimal specification for the first 256 code points, and the hexadecimal escape codes \uXXXX, \XXXXXXXX allow for the full specification of any code point. Examples of a char are 'a', '-', '\n', and '\065'.

A *string* is a sequence of characters enclosed in double quotation marks,<sup>2</sup>

· string

<sup>1</sup>Spec-4.0 p.28: char-char is missing option unicodegraph-long

<sup>2</sup>Spec-4.0 p. 28-29: simple-string-char is undefined, string-elem is unused.

| Character | Escape sequence          | Description           |
|-----------|--------------------------|-----------------------|
| BS        | \b                       | Backspace             |
| LF        | \n                       | Line feed             |
| CR        | \r                       | Carriage return       |
| HT        | \t                       | Horizontal tabulation |
| \         | \\                       | Backslash             |
| "         | \"                       | Quotation mark        |
| '         | \'                       | Apostrophe            |
| BEL       | \a                       | Bell                  |
| FF        | \f                       | Form feed             |
| VT        | \v                       | Vertical tabulation   |
|           | \uXXXX, \UXXXXXXXX, \DDD | Unicode character     |

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

```
stringChar = char - '"'
simpleString = '"' { stringChar } '"'
```

Examples are "a", "this is a string", and "-&#\@". *Newlines* and following *whitespaces*,

· newline  
· whitespace

```
newline = [CR] LF (* carriage return and line feed *)
whitespace = {SP} (* space *)
```

are taken literally, but may be ignored by a preceding \character. Further examples of strings are,

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

**Listing 5.3:** fsharp, Examples of string literals.

The response is shown in double quotation marks, which are not part of the string.

F# supports *literal types*, where the type of a literal is indicated as a prefix or suffix as shown in the Table 5.3. Examples are,

· literal type

```
> 3;;
val it : int = 3
> 4u;;
val it : uint32 = 4u
> 5.6;;
val it : float = 5.6
> 7.9f;;
val it : float32 = 7.9000001f
> 'A';;
val it : char = 'A'
> 'B'B;;
val it : byte = 66uy
> "ABC";;
```

| type            | EBNF  | Examples  |
|-----------------|---|---|
| int, int32      | (dInt   xInt) ["l"]   | 3   |
| uint32          | (dInt   xInt) ("u"   "ul")  | 3u  |
| byte, uint8     | ((dInt   xInt) "uy")   (char "B")   | 3uy   |
| byte[]          | ["@"] string "B"  | "abc"B and "@http:\\\"B"                                |
| sbyte, int8     | (dInt   xInt) "y"   | 3y  |
| int16           | (dInt   xInt) "s"   | 3s  |
| uint16          | (dInt   xInt) "us"  | 3us   |
| int64           | (dInt   xInt) "L"   | 3L  |
| uint64          | (dInt   xInt) ("UL"   "uL")   | 3UL and 3uL   |
| bignum*         | dInt "I"  | 3I  |
| nativeint       | (dInt   xInt) "n"   | 3n  |
| unativeint      | (dInt   xInt) "un"  | 3un   |
| float, double   | float   (xInt "LF")   | 3.0   |
| single, float32 | (float ("F"   "f"))   (xInt "lf")   | 3.0f  |
| decimal         | (float   dInt) ("M"   "m")  | 3.0m and 3m   |
| string          | simpleString  <br>'@' '{ (char - ('"'   '\''))   '""' } '''  <br>'''' {char} '''' (*no '""' substring*) | "a \"quote\".\n"<br>@"a "quote".\n"<br>""a "quote".\n"" |

Table 5.3: List of literal type. No spacing is allowed between the literal and the prefix or suffix. There must not be a space between the number and the literal name. The [] notation is for lists, see Chapter 8. \*bignum is not a basic type and does not yet have an implementation for dInt ("Q" | "R" | "Z" | "N" | "G") in Mono.

```
val it : string = "ABC"
```

**Listing 5.4:** fsharp, Named and implied literals.

Strings literals may be *verbatim* by the @-notation or triple double quotation marks, meaning that the escape sequences are not converted to their code point., e.g.,

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

**Listing 5.5:** fsharp, Examples of a string literal.

For strings containing double quotation marks, verbatim literals has 2 possible notations, either use the @-notation and escaping double quotation marks with an extra double quotation mark, or use triple double quotation marks. *The triple double quotation marks notation may not contain substrings that are triple double quotation marks, and thus @-notation is preferred.*

Many basic types are compatible and the type of a literal may be changed by *type casting*. E.g.,

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

**Listing 5.6:** fsharp, Casting an integer to a floating point number.

which is a `float`, since when `float` is given an argument, then it acts as a function rather than a type, and for the integer 3 it returns the floating point number 3.0. For more on functions see Chapter 6. Boolean values are often treated as the integer values 0 and 1, but no short-hand function names exists for their conversions. Instead use,

```
> System.Convert.ToBoolean 1;;
val it : bool = true
> System.Convert.ToBoolean 0;;
val it : bool = false
> System.Convert.ToInt32 true;;
```

```
val it : int = 1
> System.Convert.ToInt32 false;;
val it : int = 0
```

**Listing 5.7:** fsharp, Casting booleans.

Here `System.Convert.ToBoolean` is the identifier of a function `ToBoolean`, which is a *member* of the *class* `Convert` that is included in the *namespace* `System`. Namespaces, classes, and members are all part of Structured programming to be discussed in Part IV.

- member
- class
- namespace

Type casting is often a destructive operation, e.g., type casting a `float` to `int` removes the fractional part without rounding,

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

**Listing 5.8:** fsharp, Fractional part is removed by downcasting.

Here we type casted to a lesser type, in the sense that integers is a subset of floating point numbers, and this is called *downcasting*. The opposite is called *upcasting* and is often non-destructive, as Listing 5.6 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,

- downcasting
- upcasting
- rounding
- whole part
- fractional part

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

**Listing 5.9:** fsharp, Fractional part is removed 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.

## 5.2 Operators on basic types

Listing 5.9 is an example of an arithmetic *expression* using an *infix operator*. Expressions is the basic building block of all F# programs, and its grammar has many possible options. The grammar for expressions are defined recursively, and some of it is given by,<sup>3</sup>

- expression
- infix operator

```
bool = "true" | "false"
const = byte | sbyte | uint8 | int8 | int16 | uint16 | int | int32 | uint32 |
        int64 | uint64 | bignum | nativeint | unativeint | float | double | single |
        float32 | decimal | char | string | byte [] | bool | "()"
slice-range =
    expr (* single index *)
    | expr ".." (* from index to end *)
    | ".." expr (* from start to index *)
    | expr ".." expr (* from one index to another *)
    | "*" (* from start to end *)
expr =
    const (* constant value *)
    | "(" expr ")" (* block expression *)
    | expr operator expr (* infix operation *)
    | operator expr (* prefix operation *)
    | expr expr (* function application *)
    | expr "[" slice-range "]" (* slice lookup *)
    | ...
```

**Listing 5.10:** expressionArithmetic

<sup>3</sup>Spec-4.0 Section 4.3: `const` is missing `uint8`, `int8` `nativeint`, `unativeint`.

| Operator    | op1     | op2     | Expression         | Result       | Description                       |
|-------------|---------|---------|--------------------|--------------|-----------------------------------|
| op1 + op2   | ints    | ints    | 5 + 2              | 7            | Addition                          |
|             | floats  | floats  | 5.0 + 2.0          | 7.0          |                                   |
|             | chars   | chars   | 'a' + 'b'          | '\195'       | Addition of codes                 |
|             | strings | strings | "ab" + "cd"        | "abcd"       | Concatenation                     |
| op1 - op2   | ints    | ints    | 5 - 2              | 3            | Subtraction                       |
|             | floats  | floats  | 5.0 - 2.0          | 3.0          |                                   |
| op1 * op2   | ints    | ints    | 5 * 2              | 10           | Multiplication                    |
|             | floats  | floats  | 5.0 * 2.0          | 10.0         |                                   |
| op1 / op2   | ints    | ints    | 5 / 2              | 2            | Integer division<br>Division      |
|             | floats  | floats  | 5.0 / 2.0          | 2.5          |                                   |
| op1 % op2   | ints    | ints    | 5 % 2              | 1            | Remainder                         |
|             | floats  | floats  | 5.0 % 2.0          | 1.0          |                                   |
| op1 ** op2  | floats  | floats  | 5.0 ** 2.0         | 25.0         | Exponentiation                    |
| op1 && op2  | bool    | bool    | true && false      | false        | boolean and                       |
| op1    op2  | bool    | bool    | true    false      | false        | boolean or                        |
| op1 &&& op2 | ints    | ints    | 0b1010 &&& 0b1100  | 0b1000       | bitwise bool and                  |
| op1     op2 | ints    | ints    | 0b1010     0b1100  | 0b1110       | bitwise boolean or                |
| op1 ^^^ op2 | ints    | ints    | 0b1010 ^^^ 0b1101  | 0b0111       | bitwise boolean exclu-<br>sive or |
| op1 <<< op2 | ints    | ints    | 0b00001100uy <<< 2 | 0b00110000uy | bitwise shift left                |
| op1 >>> op2 | ints    | ints    | 0b00001100uy >>> 2 | 0b00000011uy | bitwise and                       |
| +op1        | ints    |         | +3                 | 3            | identity                          |
|             | floats  |         | +3.0               | 3.0          |                                   |
| -op1        | ints    |         | -3                 | -3           | negation                          |
|             | floats  |         | -3.0               | -3.0         |                                   |
| not op1     | bool    |         | not true           | false        | boolean negation                  |
| ~~~op1      | ints    |         | ~~~0b00001100uy    | 0b11110011uy | bitwise boolean nega-<br>tion     |

Table 5.4: Arithmetic operators on basic types. Ints, floats, chars, and strings means all built-in integer types etc.. Note that for the bitwise operations, digits 0 and 1 are taken to be `true` and `false`.

Recursion means that a rule or a function is used by the rule or function itself in its definition. See Part III for more on recursion. Infix notation means that the *operator* `op` appears between the two *operands*, and since there are 2 operands, it is a *binary operator*. As the grammar shows, the operands themselves can be expressions. Examples are `3+4` and `4+5+6`. Some operators only takes one operand, e.g., `-3`, where `-` here is used to negate a positive integer. Since the operator appears before the operand it is a *prefix operator*, and since it only takes one argument it is also a *unary operator*. Finally, some expressions are function names, which can be applied to expressions. F# supports a range of arithmetic infix and prefix operators on its built-in types shown in Table 5.4 and 5.5 and a range of mathematical functions shown in Table 5.6. Arithmetic on various types will be discussed in detail in the following sections.<sup>4</sup>

If parentheses are omitted in Listing 5.9, then F# will interpret the expression as `(int 357.6)+0.5`, which is erroneous, since addition of an integer with a float is undefined. This is an example of precedence, i.e., function evaluation takes precedence over addition meaning that it is performed before addition. Consider the arithmetic expression, whose result is bound to `a` by

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

**Listing 5.11:** fsharp, A simple arithmetic expression.

<sup>4</sup>minor comment on indexing and slice-ranges.

| Operator   | op1  | op2  | Expression   | Result   | Description           |
|------------|--|--|--|--|-----------------------|
| op1 < op2  | bool<br>ints<br>floats<br>chars<br>strings | bool<br>ints<br>floats<br>chars<br>strings | <code>true &lt; false</code><br><code>5 &lt; 2</code><br><code>5.0 &lt; 2.0</code><br><code>'a' &lt; 'b'</code><br><code>"ab" &lt; "cd"</code>                     | <code>false</code><br><code>false</code><br><code>false</code><br><code>true</code><br><code>true</code>   | Less than             |
| op1 > op2  | bool<br>ints<br>floats<br>chars<br>strings | bool<br>ints<br>floats<br>chars<br>strings | <code>true &gt; false</code><br><code>5 &gt; 2</code><br><code>5.0 &gt; 2.0</code><br><code>'a' &gt; 'b'</code><br><code>"ab" &gt; "cd"</code>                     | <code>true</code><br><code>true</code><br><code>true</code><br><code>false</code><br><code>false</code>    | Greater than          |
| op1 = op2  | bool<br>ints<br>floats<br>chars<br>strings | bool<br>ints<br>floats<br>chars<br>strings | <code>true = false</code><br><code>5 = 2</code><br><code>5.0 = 2.0</code><br><code>'a' = 'b'</code><br><code>"ab" = "cd"</code>                                    | <code>false</code><br><code>false</code><br><code>false</code><br><code>false</code><br><code>false</code> | Equal                 |
| op1 <= op2 | bool<br>ints<br>floats<br>chars<br>strings | bool<br>ints<br>floats<br>chars<br>strings | <code>true &lt;= false</code><br><code>5 &lt;= 2</code><br><code>5.0 &lt;= 2.0</code><br><code>'a' &lt;= 'b'</code><br><code>"ab" &lt;= "cd"</code>                | <code>false</code><br><code>false</code><br><code>false</code><br><code>true</code><br><code>true</code>   | Less than or equal    |
| op1 >= op2 | bool<br>ints<br>floats<br>chars<br>strings | bool<br>ints<br>floats<br>chars<br>strings | <code>true &gt;= false</code><br><code>5 &gt;= 2</code><br><code>5.0 &gt;= 2.0</code><br><code>'a' &gt;= 'b'</code><br><code>"ab" &gt;= "cd"</code>                | <code>true</code><br><code>true</code><br><code>true</code><br><code>false</code><br><code>false</code>    | Greater than or equal |
| op1 <> op2 | bool<br>ints<br>floats<br>chars<br>strings | bool<br>ints<br>floats<br>chars<br>strings | <code>true &lt;&gt; false</code><br><code>5 &lt;&gt; 2</code><br><code>5.0 &lt;&gt; 2.0</code><br><code>'a' &lt;&gt; 'b'</code><br><code>"ab" &lt;&gt; "cd"</code> | <code>true</code><br><code>true</code><br><code>true</code><br><code>true</code><br><code>true</code>      | Not Equal             |

Table 5.5: Comparison operators on basic types. Ints, floats, chars, and strings means all built-in integer types etc..

| Type                                | Function name      | Example                    | Result         | Description            |
|-------------------------------------|--------------------|----------------------------|----------------|------------------------|
| Ints and floats                     | <code>abs</code>   | <code>abs -3</code>        | 3              | Absolute value         |
| Floats                              | <code>acos</code>  | <code>acos 0.8</code>      | 0.6435011088   | Inverse cosine         |
| Floats                              | <code>asin</code>  | <code>asin 0.8</code>      | 0.927295218    | Inverse sinus          |
| Floats                              | <code>atan</code>  | <code>atan 0.8</code>      | 0.6747409422   | Inverse tangent        |
| Floats                              | <code>atan2</code> | <code>atan2 0.8 2.3</code> | 0.3347368373   | Inverse tangentvariant |
| Floats                              | <code>ceil</code>  | <code>ceil 0.8</code>      | 1.0            | Ceiling                |
| Floats                              | <code>cos</code>   | <code>cos 0.8</code>       | 0.6967067093   | Cosine                 |
| Floats                              | <code>cosh</code>  | <code>cosh 0.8</code>      | 1.337434946    | Hyperbolic cosine      |
| Floats                              | <code>exp</code>   | <code>exp 0.8</code>       | 2.225540928    | Natural exponent       |
| Floats                              | <code>floor</code> | <code>floor 0.8</code>     | 0.0            | Floor                  |
| Floats                              | <code>log</code>   | <code>log 0.8</code>       | -0.2231435513  | Natural logarithm      |
| Floats                              | <code>log10</code> | <code>log10 0.8</code>     | -0.09691001301 | Base-10 logarithm      |
| Ints, floats,<br>chars, and strings | <code>max</code>   | <code>max 3.0 4.0</code>   | 4.0            | Maximum                |
| Ints, floats,<br>chars, and strings | <code>min</code>   | <code>min 3.0 4.0</code>   | 3.0            | Minimum                |
| Ints                                | <code>pown</code>  | <code>pown 3 2</code>      | 9              | Integer exponent       |
| Floats                              | <code>round</code> | <code>round 0.8</code>     | 1.0            | Rounding               |
| Ints and floats                     | <code>sign</code>  | <code>sign -3</code>       | -1             | Sign                   |
| Floats                              | <code>sin</code>   | <code>sin 0.8</code>       | 0.7173560909   | Sinus                  |
| Floats                              | <code>sinh</code>  | <code>sinh 0.8</code>      | 0.8881059822   | Hyperbolic sinus       |
| Floats                              | <code>sqrt</code>  | <code>sqrt 0.8</code>      | 0.894427191    | Square root            |
| Floats                              | <code>tan</code>   | <code>tan 0.8</code>       | 1.029638557    | Tangent                |
| Floats                              | <code>tanh</code>  | <code>tanh 0.8</code>      | 0.6640367703   | Hyperbolic tangent     |

Table 5.6: Predefined functions for arithmetic operations

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

Table 5.7: 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. Full table is given in Table D.1.

Here, the addition and multiplication functions are shown in *infix notation* with the *operator* lexemes + and \*. To arrive at the resulting value 23, F# has to decide in which order to perform the calculation. There are 2 possible orders, `3 + (4 * 5)` or `(3 + 4) * 5`, which gives different results. For integer arithmetic, the correct order is of course to multiply before addition, and we say that multiplication takes *precedence* over addition. Every atomic operation that F# can perform is ordered in terms of its precedences, and for some common built-in operators shown in Table 5.7, 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`<sup>5</sup>

- infix notation
- operator
- precedence
- boolean or
- boolean and

```
> 3.0*4.0*5.0;;
val it : float = 60.0
> (3.0*4.0)*5.0;;
val it : float = 60.0
> 3.0*(4.0*5.0);;
val it : float = 60.0
> 4.0 ** 3.0 ** 2.0;;
val it : float = 262144.0
> (4.0 ** 3.0) ** 2.0;;
val it : float = 4096.0
> 4.0 ** (3.0 ** 2.0);;
val it : float = 262144.0
```

**Listing 5.12:** fsharp, Precedences rules define implicate parentheses.

<sup>5</sup>Spec-4.0, Table 18.2.1 appears to be missing boolean 'and' and 'or' operations. Section 4.4 seems to be missing &&& and ||| bitwise operators.



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

Table 5.8: Truth table for boolean 'and', 'or', and 'not' operators. Value 0 is false and 1 is true.

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

Advice

## 5.3 Boolean arithmetic

Boolean arithmetic is the basis of almost all computers and particularly important for controlling program flow, which will be discussed in Chapter 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 5.8. 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,

· and  
· or  
· not  
· truth table

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

val it : unit = ()
```

**Listing 5.13:** fsharp, Boolean operators and truth tables.

Spacing produced using the `printfn` function is not elegant. In Section 6.4 we will discuss better options for producing more beautiful output. Notice, that the arguments for `printfn` was given on the next line with indentation. Generally, F# ignores newlines and whitespaces except when using the lightweight syntax discussed in Chapter 6.

## 5.4 Integer arithmetic

The set of integers is infinitely large, but since all computers have limited resources, it is not possible to represent it in their entirety. The various integer types listed in Table 5.1 are finite subset reduced by limiting their ranges. Although `bigint` is theoretically unlimited, the biggest number representable is still limited by computer memory. An in-depth description of integer implementation can be found in Appendix A. The type `int` is the most common type.

Table 5.4, 5.5, and 5.6 gives examples operators and functions pre-defined for integer types. Notice that fewer functions are available for integers than for floating point numbers. For most addition, subtraction, multiplication, and negation the result straight forward. However, performing arithmetic operations on integers requires extra care, since the result since they may cause *overflow*, *underflow*, e.g., the range of the integer type `sbyte` is  $[-128 \dots 127]$ , which causes problems in the following example,

· overflow  
· underflow

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

**Listing 5.14:** fsharp, Adding integers may cause overflow.

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`,

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

**Listing 5.15:** fsharp, Subtracting integers may cause underflow.

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

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

```
> 0b10000010uy;;  
val it : byte = 130uy  
> 0b10000010y;;  
val it : sbyte = -126y
```

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

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`. Similar arguments can be made explaining underflows.

The division and remainder operators *integer division*, which discards the fractional part after division, and the *remainder* operator calculates the remainder after integer division, e.g.,

· integer division  
· remainder

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

**Listing 5.17:** fsharp, Integer division and remainder operators.

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

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

| a | b | a xor b |
|---|---|---------|
| 0 | 0 | 0       |
| 0 | 1 | 1       |
| 1 | 0 | 1       |
| 1 | 1 | 0       |

Table 5.9: Boolean exclusive or truth table.

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

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

And we see that integer division of 7 by 3 followed by multiplication by 3 is less than 7, and the difference is 7 % 3.

Notice that neither overflow nor underflow error gave rise to an error message, which is why such bugs are difficult to find. Dividing any non-zero number with 0 is infinite, which is also outside the domain of any of the integer types, but in this case, F# casts an *exception*,

```
> 3/0;;
System.DivideByZeroException: Attempted to divide by zero.
  at <StartupCode$FSI_0002>.$FSI_0002.main@ () <0x68079f8 + 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) <0x1a7c270 + 0x000a1> in <
    filename unknown>:0
Stopped due to error
```

**Listing 5.19:** fsharp, Integer division by zero causes an exception run-time error.

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

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

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

**Listing 5.20:** fsharp, Integer exponent function.

which is equal to  $2^5$ .

For binary arithmetic on integers, the following operators are available: `op1 <<< op2`, which shifts the bit pattern of `op1` `op2` positions to the left insert 0's to right; `op1 >>> op2`, which shifts the bit pattern of `op1` `op2` positions to the right insert 0's to left; `op1 &&& op2`, Bitwise 'and', returns the result of taking the boolean 'and' operator position-wise; `op1 ||| op2`, Bitwise 'or', as 'and' but using the boolean 'or' operator; and `op1 ~~~ op2`, Bitwise xor, which returns the result of the boolean 'xor' operator defined by the truth table in Table 5.9.

## 5.5 Floating point arithmetic

The set of reals is infinitely large, and since all computers have limited resources, it is not possible to represent it in their entirety. The various floating point types listed in Table 5.1 are finite subset

reduced by sampling the space of reals. An in-depth description of floating point implementations can be found in Appendix A. The type `float` is the most common type.

Table 5.4, 5.5, and 5.6 gives examples operators and functions pre-defined for floating point types. For most addition, subtraction, multiplication, divisions, and negation the result straight forward.

The remainder operator for floats calculates the remainder after division and discarding the fractional part,

```
> 7.0 / 2.5;;  
val it : float = 2.8  
> 7.0 % 2.5;;  
val it : float = 2.0
```

**Listing 5.21:** fsharp, Floating point division and remainder operators.

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

```
> float (int (7.0 / 2.5));;  
val it : float = 2.0  
> (float (int (7.0 / 2.5))) * 2.5;;  
val it : float = 5.0  
> (float (int (7.0 / 2.5))) * 2.5 + 7.0 % 2.5;;  
val it : float = 7.0
```

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

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.,

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

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

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.,

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

**Listing 5.24:** fsharp, Floating point arithmetic has finite precision.

That is, addition and subtraction associates to the left, hence the expression is interpreted as  $(357.8 + 0.1) - 357.9$ , and we see that we do not get the expected 0, since only a limited number of floating point values are available, and the numbers  $357.8 + 0.1$  and  $357.9$  do not result in the same floating point representation. Such errors tend to accumulate and comparing the result of expressions of floating point values should therefore be treated with care. Thus, *equivalence of two floating point expressions should only be considered up to sufficient precision, e.g., comparing  $357.8 + 0.1$  and  $357.9$  up to  $1e-10$  precision should be tested as,  $\text{abs}((357.8 + 0.1) - 357.9) < 1e-10$ .*

Advice

## 5.6 Char and string arithmetic

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

· ASCIIbetical order

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

**Listing 5.25:** fsharp, Converting case by casting and integer arithmetic.

I.e., the code point difference between upper and lower case for any alphabetical character 'a' to 'z' is constant, hence we can change case by adding or subtracting the difference between any corresponding character. Unfortunately, this does not generalize to characters from other languages.

A large collection of operators and functions exist for `string`. The most simple is concatenation using, e.g.,

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

**Listing 5.26:** fsharp, Example of string concatenation.

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

```
> "abcdefg".[0];;
val it : char = 'a'
> "abcdefg".[3];;
val it : char = 'd'
> "abcdefg".[3..];;
val it : string = "defg"
> "abcdefg".[..3];;
val it : string = "abcd"
> "abcdefg".[1..3];;
val it : string = "bcd"
> "abcdefg".[*];;
val it : string = "abcdefg"
```

**Listing 5.27:** fsharp, String indexing using square brackets.

Notice, that the first character has index 0, and to get the last character in a string, we use the string's length property as,

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

**Listing 5.28:** fsharp, String length attribute and string indexing.

Notice, since index counting starts at 0, and the string length is 7, then the index of the last character is 6. An alternative notation for indexing is to use the property `Char`, and in the example `'abcdefg'. [3]` is the same as `a.Char 3`. There is a long list of built-in functions in `System.String` for working with strings, some of which will be discussed in Chapter 13.1.

The *dot notation* is an example of Structured programming, where technically speaking, the string `"abcdefg"` is an immutable *object* of class `string`, and `[]` is an object *method* and `Length` is a property. For more on object, classes, and methods see Chapter 18.

Strings are compared letter by letter. For two strings to be equal, they must have the same length and all the letters must be identical. E.g., `"abs" = "absalon"` is false, while `"abs" = "abs"` is true. The `<>` operator is the boolean negation of the `=` operator, e.g., `"abs" <> "absalon"` is true, while `"abs" <> "abs"` is false. For the `<`, `<=`, `>`, and `>=` operators, the strings are ordered alphabetically, such that `"abs" < "absalon" && "absalon" < "milk"` is true, that is, the `<` operator on two strings is true, if the left operand should come before the right, when sorting alphabetically. The algorithm for deciding the boolean value of `lOp < rOp` is as follows: we start by examining the first character, and if `lOp.[0]` and `rOp.[0]` are different, then the `lOp < rOp` is equal to `lOp.[0] < rOp.[0]`. E.g.,

· . []

· dot notation  
· object  
· class  
· method

`"milk" < "abs"` is the same as `'m' < 'a'` is false, since the letter 'm' does not come before the letter 'a' in the alphabet, or more precisely, the codepoint of 'm' is not less than the codepoint of 'a'. If `lop.[0]` and `rop.[0]` are equal, then we move onto the next letter and repeat the investigation, e.g., `"abe" < "abs"` is true, since `"ab" = "ab"` is true and `'e' < 's'` is true. If we reach the end of either of the two strings, then the short is smaller than the larger, e.g., `"abs" < "absalon"` is true, while `lstinline!"abs" < "abs"!` is false. The `<=`, `>`, and `>=` operators are defined similarly.

## Chapter 6

# Constants, functions, and variables

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

$$ax^2 + bx + c = 0 \quad (6.1)$$

we use the quadratic formula from elementary algebra,

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

and write a small program that defines functions calculating relevant values for any set of coefficients,

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

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

```
1.0 * x ** 2.0 + 0.0 * x + -1.0
    has determinant 2.0 and solutions -0.7071067812 and 0.7071067812
```

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

Here 3 functions are defined as `determinant`, `positiveSolution`, and `negativeSolution` are defined, and applied to 3 values named `a`, `b`, and `c`, and the results are named `d`, `xn`, and `xp`. These names are examples of identifiers, and with these, we may reuse the quadratic formulas and calculated values later, while avoiding possible typing mistakes and reducing amount of code, which needs to be debugged.

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

· lightweight  
syntax

The use of identifiers is central in programming. For F# not to be confused by built-in functionality, identifiers must follow a specific grammar: An identifier must start with a letter, but can be followed by zero or more of letters, digits, and a range of special characters except SP, LF, and CR (space,

abstract, and, as, assert, base, begin, class, default, delegate, do, done, downcast, downto, elif, else, end, exception, extern, false, finally, for, fun, function, global, if, in, inherit, inline, interface, internal, lazy, let, match, member, module, mutable, namespace, new, null, of, open, or, override, private, public, rec, return, sig, static, struct, then, to, true, try, type, upcast, use, val, void, when, while, with, and yield.

Figure 6.1: List of keywords in F#.

atomic, break, checked, component, const, constraint, constructor, continue, eager, fixed, fori, functor, include, measure, method, mixin, object, parallel, params, process, protected, pure, recursive, sealed, tailcall, trait, virtual, and volatile.

Figure 6.2: List of reserved keywords for possible future use in F#.

line feed, and carriage return). An identifier must not be a keyword or a reserved-keyword listed in Figures 6.1, 6.2, 6.3, and 6.4

For characters in the Basic Latin Block, i.e., the first 128 code points alias ASCII characters, an ident is,

```
dDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
letter = "A" | "B" | ... | "Z" | "a" | "b" | ... | "z"
specialChar = "_"
ident = (letter | "_") {letter | dDigit | specialChar}
```

Thus, examples of identifiers are `a`, `theCharacter9`, `Next_Word`, `_tok`. The for the full definition, `letter = Lu | Ll | Lt | Lm | Lo | Nl` and `specialChar = Pc | Mn | Mc | Cf`, which refers to the Unicode general categories described in Appendix B.3, and there are currently 19,345 possible Unicode code points in the `letter` category and 2,245 possible Unicode code points in the `specialChar` category. Binding expressions to identifiers is done with the keyword `let`, using the following simplified syntax:

```
arg = ident | "(" ident ":" type ")"
argList = arg | arg argList
identOrOp = ident | ( operatorName )
valueDefn = ["mutable " ] ident [ ":" type ] "=" expr
functionDefn = ident-or-op argList [ ":" type ] "=" expr "
expr = ...
| "let" valueDefn "in" expr (* binding a value or variable *)
| "let" functionDefn "in" expr (* binding a function or operator *)
| "fun" argList "->" expr (* a function as value *)
| expr ":" type (* type annotation *)
| "begin" expr "end" (* alternative block expression *)
| expr; expr (* sequence of expression *)
| ...
```

which will be discussed in the following.<sup>1</sup>

<sup>1</sup>Spec-4.0 Section 6.6, function-or-value-defns, possible Mono deviation from specification: `let rec functionDefn` and `functionDefn` requires newline before and.

`let!`, `use!`, `do!`, `yield!`, `return!`, `|`, `->`, `<-`, `..`, `:`, `(`, `)`, `[`, `]`, `[<`, `>]`, `[|`, `|]`, `{`, `}`, `'`, `#`, `!?`, `?:`, `:`, `>`, `...`, `::`, `:=`, `;;`, `;`, `=`, `_`, `?`, `??`, `(*)`, `<@`, `@>`, `<@@`, and `@@>`.

Figure 6.3: List of symbolic keywords in F#.



~ and `.

Figure 6.4: List of reserved symbolic keywords for possible future use F#.

## 6.1 Values

Binding identifiers to literals or expressions that are evaluated to be values is called value binding, and examples are `let a = 3.0` and `let b = cos 0.9`. On EBNF the simplified syntax,

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

I.e., the `let` keyword dictates that the identifier `ident` is an alias of the expression `expr`. The type may be specified with the `:` lexeme to type `type`. The binding may be mutable as indicated by the keyword `mutable`, which will be discussed in Section 6.5, and the binding holds *lexically* for the last expression as indicated by the `in` keyword.<sup>2</sup> For example, letting the identifier `p` be bound to the value 2.0 and using it in an expression is done as follows,

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

```
8.0
```

**Listing 6.2:** `letValue.fsx` - The identifier `p` is used in the expression following the `in` keyword.

In the interactive mode used in the example above, we see that F# infers the type... F# will ignore most newlines between lexemes, i.e., the above is equivalent to writing,

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

```
9.0
```

**Listing 6.3:** `letValueLF.fsx` - Newlines after `in` make the program easier to read.

F# also allows for an alternative notation called *lightweight syntax*, where e.g., the `in` keyword is replaced with a newline, and the expression starts on the next line at the same column as `let` starts in, i.e., the above is equivalent to

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

```
9.0
```

**Listing 6.4:** `letValueLightWeight.fsx` - Lightweight syntax does not require the `in` keyword, but expression must be aligned with the `let` keyword.

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

```
> let p = 2.0
- printfn "%A" (3.0 ** p);;
9.0

val p : float = 2.0
val it : unit = ()
```

**Listing 6.5:** `fsharp`i, Interactive mode also responds inferred types.

<sup>2</sup><https://coders-corner.net/2013/11/12/lexical-scope-vs-dynamic-scope/>

By the `val` keyword in the line `val p : float = 2.0` we see that `p` is inferred to be of type `float` and bound to the value 2.0. The inference is based on the type of the right-hand-side, which is of type `float`. Identifiers may be defined to have a type using the `:` lexeme, but the types on the left-hand-side and right-hand-side of the `=` lexeme must be identical. I.e., mixing types gives an error,

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

```
/Users/sporring/repositories/fsharpNotes/src/letValueTypeError.fsx(1,17):
    error FS0001: This expression was expected to have type
        float
but here has type
    int
```

**Listing 6.6:** `letValueTypeError.fsx` - Binding error due to type mismatch.

Here, the left-hand-side is defined to be an identifier of type `float`, while the right-hand-side is a literal of type integer.

An expression can be a sequence of expressions separated by the lexeme `;`, e.g.,

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

```
2.0
9.0
```

**Listing 6.7:** `letValueSequence.fsx` - A value binding for a sequence of expressions.

The lightweight syntax automatically inserts the `;` lexeme at newlines, hence using the lightweight syntax the above is the same as,

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

```
2.0
9.0
```

**Listing 6.8:** `letValueSequenceLightWeight.fsx` - A value binding for a sequence using lightweight syntax.

A key concept of programming is *scope*. In F#, the scope of a value binding is lexically meaning that the binding is constant from the `let` statement defining it, until it is redefined, e.g.,

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

```
4
```

**Listing 6.9:** `letValueScopeLower.fsx` - Redefining identifiers is allowed in lower scopes.

Scopes are given levels, and scopes may be nested, where the nested scope has a level one lower than its parent.<sup>3</sup> F# distinguishes between the top and lower levels, and at the top level in the lightweight syntax, redefining values is not allowed, e.g.,

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

<sup>3</sup>Drawings would be good to describe scope

```
/Users/sporring/repositories/fsharpNotes/src/letValueScopeLowerError.fsx(2,5):
error FS0037: Duplicate definition of value 'p'
```

Listing 6.10: letValueScopeLowerError.fsx - Redefining identifiers is not allowed in lightweight syntax at top level.

But using `begin` and `end` keywords, we create a *block* which acts as a *nested scope*, and then redefining is allowed, e.g.,

· block  
· nested scope

```
begin
  let p = 3
  let p = 4
  printfn "%A" p
end
```

4

Listing 6.11: letValueScopeBlockAlternative2.fsx - A block has lower scope level, and rebinding is allowed.

It is said that the second binding *overshadows* the first. Alternatively we may use parentheses to create a block, e.g.,

· overshadows

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

4

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

In both cases we used indentation, which is good practice, but not required here. Lowering level is a natural part of function definitions to be discussed in Section 6.2 and flow control structures to be discussed in Chapter 7.

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

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

4  
4

**Listing 6.13:** letValueScopeBlockProblem.fsx - Overshadowing hides the first binding.

will print the value 4 last bound to the identifier `p`, since lexeme `;` associates to the right, i.e., the above is interpreted as `let p = 3 in let p = 4 in (printfn "%A"p; printfn "%A"p)`. Instead we may create a block as,<sup>4</sup>

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

4  
3

**Listing 6.14:** letValueScopeBlock.fsx - Blocks allow for the return to the previous scope.

<sup>4</sup>spacing in linline mode after double quotation mark is weird.

Here the lexical scope of `let p = 4 in ...` is for the nested scope, which ends at `)`, returning to the lexical scope of `let p = 3 in ....` Alternatively, the `begin` and `end` keywords could equally have been used.

5

## 6.2 Non-recursive functions

A function is a mapping between an input and output domain. A key advantage of using functions, when programming, is that they *encapsulate code* into smaller units, that are easier to debug and may be reused. F# is a functional first programming language, and offers a number of alternative methods for specifying parameters, which will be discussed in this section. Binding identifiers to functions follows a syntax similar to value binding,

· encapsulate  
code

```
arg = ident | "(" ident ":" type ")"
argList = arg | arg argList
identOrOp = ident | ( operatorName )
functionDefn = identOrOp argList [":" type] "=" expr
expr = ...
    | "let" functionDefn "in" expr (* binding a function or operator *)
    | ...
```

Functions may also be recursive, which will be discussed in Chapter 7. An example in interactive mode is,

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

val sum : x:float -> y:float -> float
val c : float = 1221.0
val it : unit = ()
```

**Listing 6.15:** fsharp, An example of a binding of an identifier and a function.

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

Not all types need to be declared, just sufficient for F# to be able to infer the types for the full statement. In the example, one sufficient specification is, and we could just have specified the type of the result,

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

**Listing 6.16:** All types need most often not be specified.

or even just one of the arguments,

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

**Listing 6.17:** Just one type is often enough for F# to infer the rest.

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

· operator  
· operand

<sup>5</sup>Remember to say something about interactive scripts and the `;;` lexeme and scope

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

```
1221.0
```

**Listing 6.18:** letFunctionLightWeight.fsx - Lightweight syntax for function definitions.

Arguments need not always be inferred to types, but may be of generic type, which F# prefers, when *type safety* is ensured, e.g.,

· type safety

```
> 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

val second : x:'a -> y:'b -> 'b
val a : int = 5
val b : float = 5.0
val it : unit = ()
```

Listing 6.19: fsharpi, Typesafety implies that a function will work for any type, and hence it is generic.

Here the function `second` does not use the first argument, `x` which is any type called `'a`, and the type of the second element, `y`, is also any type and not necessarily the same as `x`, so it is called `'b`. Finally the result is the same type as `y`, whatever it is. This is an example of a *generic function*.

· generic function

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

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

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

```
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
has solutions -0.7071067812 and 0.7071067812
```

**Listing 6.20:** identifiersExampleAdvance.fsx - A function may contain sequences of expressions.

Here we used the lightweight syntax, where the `=` identifies the start of a nested scope, and F# identifies the scope by indentation. The amount of space used for indentation is does not matter, but all lines following the first must use the same. The scope ends before the first line with the previous indentation or none. Notice how the last expression is not bound to an identifier, but is the result of the function, i.e., in contrast to many other languages, F# does not have an explicit keyword for returning values. Note also that since the function `determinant` is defined in the nested scope of `solution`, then `determinant` cannot be called outside `solution`, since the scope ends before `let a = 1.0`.

*Lexical scope* and function definitions can be a cause of confusion as the following example shows,<sup>6</sup>

· lexical scope

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

6.0

Listing 6.21: lexicalScopeNFunction.fsx - Lexical scope means that  $f(z) = 3x$  and not  $4x$  at the time of calling.

Here the value binding for `a` is redefined, after it has been used to define a helper function `f`. So which value of `a` is used when we later apply `f` to an argument? To resolve the confusion, remember that value binding is lexically defined, i.e., the binding `let f z = a * x` uses the value of `a`, it has by the ordering of the lines in the script, not dynamically by when `f` was called. Hence, *think of lexical scope as substitution of an identifier with its value or function immediately at the place of definition*. I.e., since `a` and `3.0` are synonymous in the first lines of the program, then the function `f` is really defined as, `let f z = 3.0 * x`.<sup>7</sup>

Advice

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

· anonymous function

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

5

**Listing 6.22:** functionDeclarationAnonymous.fsx - Anonymous functions are functions as values.

Here a name is bound to an anonymous function, which returns the first of two arguments. The difference to `let first x y = x` is that anonymous functions may be treated as values, meaning that they may be used as arguments to other functions, and new values may be reassigned to their identifiers, when mutable, as will be discussed in Section 6.5. A common use of anonymous functions is as arguments to other functions, e.g.,

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

18

Listing 6.23: functionDeclarationAnonymousAdvanced.fsx - Anonymous functions are often used as arguments for other functions.

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

Advice

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)
```

<sup>6</sup>Add a drawing or possibly a spell-out of lexical scope here.

<sup>7</sup>comment on dynamic scope and mutable variables.

```
15
6
```

**Listing 6.24:** 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 6.25:** 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.* Advice

A *procedure* is a generalisation of the concept of functions, and in contrast to functions procedures need not return values, · procedure

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

```
This is '3'
This is '3.0'
```

Listing 6.26: procedure.fsx - A procedure is a function that has no return value, which in F# implies () as return value.

In F# this is automatically given the unit type as return value. Procedural thinking is useful for *encapsulation* of scripts, but is prone to *side-effects* and should be minimized by being replaced by functional thinking. More on side-effects in Section 6.5. 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. · encapsulation · side-effects

8

## 6.3 User-defined operators

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

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

```
3.0 plus 4.0 is 7.0 and 7.0
```

**Listing 6.27:** addOperatorNFunction.fsx -

---

<sup>8</sup>Remember examples of return of functions, no arguments (), and wildcard patterns as arguments \_.

All operator has this option, and you may redefine them and define your own operators, who has names specified by the following simplified EBNF:

```
infixOrPrefixOp := "+" | "-" | "+. " | "-. " | "%" | "&" | "&&"
tildes = "~" | "~" tildes
prefixOp = infixOrPrefixOp | tildes | (! {opChar} - "!=")
dots = "." | "." dots
infixOp =
  {dots} (
    infixOrPrefixOp
    | "-" {opChar}
    | "+" {opChar}
    | "||"
    | "<" {opChar}
    | ">" {opChar}
    | "="
    | " |" {opChar}
    | "&" {opChar}
    | "^" {opChar}
    | "*" {opChar}
    | "/" {opChar}
    | "%" {opChar}
    | "!=" )
  | ":@" | ":@" | "$" | "?"
```

**Listing 6.28:** Grammar for infix and prefix lexemes

The precedence rules and associativity of user-defined operators follows the rules for which they share prefixes with built-in rules, see Table D.1. E.g., `.*`, `+++`, and `<+` are valid operator names for infix operators, they have precedence as ordered, and their associativity are all left. Using `~` as the first character in the definition of an operator makes the operator unary and will not be part of the name. Examples of definitions and use of operators are,

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

```
13
16
true
2
```

**Listing 6.29:** operatorDefinitions.fsx -

Beware, redefining existing operators lexically redefines all future uses of operator for all types, hence *it is not a good idea to redefine operators, but better to define new*.<sup>9</sup> In Chapter /refchap:oop we will

Advice

discuss how to define type specific operators including prefix operators. Operators beginning with `*` must use a space in its definition, ( `*` in order for it not to be confused with the beginning of a comment (`*`.<sup>10</sup>

<sup>9</sup>It seems there is a bug in mono: `let ( +) x = x+1 in printfn "%A" +1;;` prints 1 and not 2.

<sup>10</sup>this requires comments to be describe previously!



| Placeholder                         | Type  | Description  |
|-------------------------------------|---|--|
| <code>%b</code>                     | <code>bool</code>                                     | Replaces with boolean value  |
| <code>%s</code>                     | <code>string</code>                                   |  |
| <code>%c</code>                     | <code>char</code>                                     |  |
| <code>%d, %i</code>                 | basic integer   |  |
| <code>%u</code>                     | basic unsigned integers                               |  |
| <code>%x</code>                     | basic integer   | formatted as unsigned hexadecimal with lower case letters                                      |
| <code>%X</code>                     | basic integer   | formatted as unsigned hexadecimal with upper case letters                                      |
| floating point type <code>%o</code> | basic integer   | formatted as unsigned octal integer  |
| <code>%f, %F,</code>                | basic floats  | formatted on decimal form  |
| <code>%e, %E,</code>                | basic floats  | formatted on scientific form. Lower case uses "e" while upper case uses "E" in the formatting. |
| <code>%g, %G,</code>                | basic floats  | formatted on the shortest of the corresponding decimal or scientific form.                     |
| <code>%M</code>                     | decimal   |  |
| <code>%O</code>                     | Objects <code>ToString</code> method                  |  |
| <code>%A</code>                     | any built-in types                                    | Formatted as a literal type  |
| <code>%a</code>                     | <code>Printf.TextWriterFormat -&gt;'a -&gt; ()</code> |  |
| <code>%t</code>                     | <code>(Printf.TextWriterFormat -&gt; ()</code>        |  |

Table 6.1: Printf placeholder string

## 6.4 Printf

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

· `printf`

```
"printf" formatString {ident}
```

where a `formatString` is a string (simple or verbatim) with placeholders,

```
placeholder = "%%" | ""% "0" ["+"] ["-"] [SP] [dInt] [ "." dInt] [
    placeholderType]
placeholderType = "b" | "d" | "i" | "u" | "x" | "X" | "o" | "e" | "E" | "f" | "F"
    | "g" | "G" | "M" | "O" | "A" | "a" | "t"
```

and where the number of arguments after `formatString` must match the number of placeholders in `formatString`. The `placeholderType` is elaborated in Table 6.1. The function `printf` prints `formatString` to the console, where all placeholder has been replaced by the value of the corresponding argument formatted as specified. E.g.,

```
let pi = 3.1415192
```

```

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

```

```

An integer: 3
A char 'h' and a string "hello"
Float using width 8 and 1 number after the decimal:
  "   3.1" "  -3.1"
  "000003.1" "-00003.1"
  "   3.1" "  -3.1"
  "3.1" " -3.1"
  "  +3.1" " -3.1"
  "   hello"
"hello"

```

**Listing 6.30:** printfExample.fsx - Examples of printf and some of its formatting options.

Not all combinations of flags and identifier types are supported, e.g., strings cannot have number of integers after the decimal specified.<sup>11 12</sup> The placeholder types "%A", "%a", and "%t" are special for F#, examples of their use are,

```

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

```

```

Print examples: 3.0M, 3uy, "a string"
Print function with no arguments: I will not print anything
Print function with 1 argument: Custom formatter got: "3.0"

```

**Listing 6.31:** printfExampleAdvance.fsx -

The %A is special in that all built-in types including tuples, lists, and arrays to be discussed in Chapter 8 can be printed using this formatting string, but notice that the formatting performed includes the named literal string. The two formatting strings %t and %a are options for user-customizing the formatting, and will not be discussed further.

Beware, formatString is not a string but a Printf.TextWriterFormat, so let str = "hello %s" in printf str "world" will be a type error.

The family of printf is shown in Table 6.2. The function fprintf prints to a stream, e.g., stderr and stdout, of type System.IO.TextWriter. Streams will be discussed in further detail in Chapter 11. The function failwithf is used with exceptions, see Chapter 9 for more details. The function has a number of possible return value types, and for testing the ignore function ignores it all, e.g., ignore (failwithf "%d failed apples" 3)<sup>13</sup>

<sup>11</sup>Mono seems to have a bug, printfn "%.1g" 3.13;; and printfn "%.1f" 3.13;; produces different number of digits.

<sup>12</sup>Spec-4.0 %s and %b are missing in Section 3.1.16.

<sup>13</sup>Mono: bprintf and kprintf are undefined.

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

Table 6.2: The family of printf functions.

## 6.5 Variables

The `mutable` in `let` bindings means that the identifier may be rebound to a new value using the lexeme, e.g.,<sup>14</sup>

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

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

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

```
5
-3
```

Listing 6.32: `mutableAssignReassignShort.fsx` - A variable is defined and later reassigned a new value.

Here a area in memory was denoted `x`, initially assigned the integer value 5, hence the type was inferred to be `int`. Later, this value of `x` was replaced with another integer using the `<-` lexeme. The `<-` lexeme is used to distinguish the assignment from the comparison operator, i.e., if we by mistake had written,

```
> let mutable a = 0
- a = 3;;

val mutable a : int = 0
val it : bool = false
```

Listing 6.33: `fsharp`, Common error - mistaking `=` and `<-` lexemes for mutable variables. The former is the test operator, while the latter is the assignment expression.

then we instead would have obtained the default assignment of the result of the comparison of the content of `a` with the integer 3, which is `false`. However, it's important to note, that when the variable is initially defined, then the `'='` operator must be used, while later reassignments must use the `<-` expression.

Assignment type mismatches will result in an error,

```
let mutable x = 5
printfn "%d" x
```

<sup>14</sup>Discussion on heap and stack should be added here.

```
x <- -3.0
printfn "%d" x
```

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

Listing 6.34: mutableAssignReassingTypeError.fsx - Assignment type mismatching causes a compile time error.

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

A typical variable is a counter of type integer, and a typical use of counters is to increment them, 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 6.35: mutableAssignIncrement.fsx - Variable increment is a common use of variables.

A 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 6.36: mutableAssignIncrementEncapsulation.fsx -

<sup>15</sup> 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.

Variables implement dynamic scope, e.g., in comparison with the lexical scope, where the value of an identifier depends on which line in the program, an identifier is defined, dynamic scope depends on, when it is used. E.g., the script in Listing 6.21 defines a function using lexical scope and returns the number 6.0, however, if `a` is made `mutable`, then the behaviour is different:

```
let testScope x =
    let mutable a = 3.0
    let f z = a * x
```

<sup>15</sup> Explain why this works!

· encapsulation

```

a <- 4.0
f x
printfn "%A" (testScope 2.0)

```

```
8.0
```

Listing 6.37: `dynamicScopeNFunction.fsx` - Mutual variables implement dynamics scope rules. Compare with Listing 6.21.

Here the respons is 8.0, since the value of `a` changed before the function `f` was called. Variables cannot be returned from functions, that is,

```

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

```

```
0
```

**Listing 6.38:** `mutableAssignReturnValue.fsx` -

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

```

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

```

```

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

```

**Listing 6.39:** `mutableAssignReturnVariable.fsx` -

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

```

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

```

```
0
```

```
3
```

**Listing 6.40:** `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,<sup>16</sup>

<sup>16</sup>Discuss side-effects!

```

let updateFactor factor =
    factor := 2

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

printfn "%d" (multiplyWithFactor 3)

```

6

**Listing 6.41:** 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 6.42:** 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.

17

## 6.6 In-code documentation

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

1. Communicate what the code should be doing
2. Highlight big insights essential for the code
3. Highlight possible conflicts and/or areas, where the code could be changed later

The essential point is that coding is a journey in problem solving, and proper documentation is an aid in understanding the solution and the journey leading to it. Documentation is most often a mixture between in-code documentation and accompanying documents. Here we will focus on in-code documentation, but arguably this does cause problems in multi-language environments, and run the risk of bloating code.

F# has the following simplified syntax for in-code documentation,

---

<sup>17</sup>Add something about mutable functions

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

Table 6.3: Recommended XML tags for documentation comments, from ECMA-334 3rd Edition, Annex E, Section 2.

```
blockComment = "(*" {codePoint} "*)"
lineComment = "//" {codePoint - newline} newline
```

That is, text framed as a `blockComment` is still parsed by F# as keywords and basic types implying that `(* a comment (* in a comment *))` and `(* "*)" *)` are valid comments, while `(* " *)` is invalid.<sup>18</sup>

The F# compiler has an option for generating *Extensible Markup Language (XML)* files from scripts using the C# documentation comments tags<sup>19</sup>. The XML documentation starts with a triple-slash `///`, i.e., a `lineComment` and a slash, which serves as comments for the code construct, that follows immediately after. XML consists of tags which always appears in pairs, e.g., the tag “tag” would look like `<tag> ... </tag>`. The F# accept any tags, but recommends those listed in Table 6.3. If no tags are used, then it is automatically assumed to be a `<summary>`. An example of a documented script is,

· Extensible  
Markup  
Language  
· XML

```
/// Calculate the determinant of a quadratic equation with parameters a, b,
    and c
let determinant a b c =
    b ** 2.0 - 2.0 * a * c

/// <summary>Find x when 0 = ax^2+bx+c.</summary>
/// <remarks>Negative determinants are not checked.</remarks>
/// <example>
///     The following code:
///     <code>
///         let a = 1.0
///         let b = 0.0
///         let c = -1.0
///         let xp = (solution a b c +1.0)
///         printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
///     </code>
```

<sup>18</sup>Listing colors is bad.

<sup>19</sup>For specification of C# documentations comments see ECMA-334 3rd Edition, Annex E, Section 2: <http://www.ecma-international.org/publications/files/ECMA-ST/ECma-334.pdf>

```

///      results in <c>0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7</c> printed to the
      console.
/// </example>
/// <param name="a">Quadratic coefficient.</param>
/// <param name="b">Linear coefficient.</param>
/// <param name="c">Constant coefficient.</param>
/// <param name="sgn">+1 or -1 indicating which solution is to be calculated
      .</param>
/// <returns>The solution to x.</returns>
let solution a b c sgn =
    let d = determinant a b c
    (-b + sgn * sqrt d) / (2.0 * a)

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

```

```
0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7
```

**Listing 6.43:** commentExample.fsx - Code with XML comments.

Mono's `fsharpc` command may be used to extract the comments into an XML file,

```

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

```

This results in an XML file with the following content,

```

<?xml version="1.0" encoding="utf-8"?>
<doc>
<assembly><name>commentExample</name></assembly>
<members>
<member name="M:CommentExample.solution(System.Double,System.Double,System.
    Double,System.Double)">
    <summary>Find x when 0 = ax^2+bx+c.</summary>
    <remarks>Negative determinants are not checked.</remarks>
    <example>
        The following code:
        <code>
            let a = 1.0
            let b = 0.0
            let c = -1.0
            let xp = (solution a b c +1.0)
            printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
        </code>
        results in <c>0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7</c> printed to the
            console.
    </example>
    <param name="a">Quadratic coefficient.</param>
    <param name="b">Linear coefficient.</param>
    <param name="c">Constant coefficient.</param>
    <param name="sgn">+1 or -1 indicating which solution is to be calculated.</
        param>
    <returns>The solution to x.</returns>
</member>
<member name="M:CommentExample.determinant(System.Double,System.Double,System.
    Double)">

```



```
<summary>
  Calculate the determinant of a quadratic equation with parameters a, b, and c
</summary>
</member>
</members>
</doc>
```

The extracted XML is written in C# type by convention, since F# is part of the Mono and .Net framework that may be used by any of the languages using Assemblies. Besides the XML inserted in the script, the XML has added `<?xml ...>` header, `<doc>`, `<assembly>`, `<members>`, and `<member>` tags. The header and the `<doc>` tag are standards for XML. The extracted XML is geared towards documenting big libraries of codes and thus highlights the structured programming organization, see Part IV, and `<assembly>`, `<members>`, and `<member>` are indications for where the functions belong in the hierarchy. As an example, the prefix `M:CommentExample.` means that it is a method in the namespace `CommentExample`, which in this case is the name of the file. Further, the function type `val solution : a:float -> b:float -> c:float -> sgn:float -> float` is in the XML documentation `M:CommentExample.solution(System.Double,System.Double,System.Double,System.Double)`, which is the C# equivalent.

An accompanying program in the Mono suite is `mdoc`, which primary use is to perform a syntax analysis of an assembly and generate a scaffold XML structure for an accompanying document. With the `-i` flag, it is further possible to include the in-code comments as initial descriptions in the XML. The XML may be updated gracefully by `mdoc` as the code develops, without destroying manually entered documentation in the accompanying documentation. Finally, the XML may be exported to HTML. The primary use of the `mdoc` command is to analyze compiled code and generate an empty XML structure with placeholders to describe functions, values, and variables. This structure can then be updated and edited as the program develops. The edited XML files can then be exported to *Hyper Text Markup Language* (HTML) files, which can be viewed in any browser. Using the console, all of this is accomplished by,

```
$ mdoc update -o commentExample -i commentExample.xml commentExample.exe
New Type: CommentExample
Member Added: public static double determinant (double a, double b, double c);
Member Added: public static double solution (double a, double b, double c,
double sgn);
Member Added: public static double a { get; }
Member Added: public static double b { get; }
Member Added: public static double c { get; }
Member Added: public static double xp { get; }
Namespace Directory Created:
New Namespace File:
Members Added: 6, Members Deleted: 0
$ mdoc export-html -out commentExampleHTML commentExample
.CommentExample
```

The primary use of the `mdoc` command is to analyze compiled code and generate an empty XML structure with placeholders to describe functions, values, and variables. This structure can then be updated and edited as the program develops. The edited XML files can then be exported to HTML files, which can be viewed in any browser, an example of which is shown in Figure 6.5. A full description of how to use `mdoc` is found here<sup>20</sup>.

· Hyper Text  
Markup  
Language  
· HTML

<sup>20</sup><http://www.mono-project.com/docs/tools+libraries/tools/monodoc/generating-documentation/>

## solution Method

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

## Syntax

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

## Parameters

*a*  
Quadratic coefficient.

*b*  
Linear coefficient.

*c*  
Constant coefficient.

*sgn*  
+1 or -1 indicating which solution is to be calculated.

## Returns

The solution to  $x$ .

## Remarks

Negative determinants are not checked.

## Example

The following code:

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

results in  $0 = 1.0x^2 + 0.0x + -1.0 \Rightarrow x_+ = 0.7$  printed to the console.

## Requirements

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

Figure 6.5: Part of the HTML documentation as produce by `mdoc` and viewed in a browser.

## Chapter 7

# Controlling program flow

Non-recursive functions encapsulates code and allows for some control of flow, that is, if there is a piece of code, which we need to have executed many times, then we can encapsulate it in the body of a function, and then call the function several times. In this chapter, we will look at more general control of flow via loops, conditional execution, and recursion, and therefore we look at further extension of the `expr` rule,

```
pat = const | ...
guard = "when" expr
rule = pat [guard] -> expr
rules = "|" rule | "|" rule rules (* first '|' is optional *)
expr = ...
  | "for" pat "in" expr "do" expr ["done"] (* for expression *)
  | "for" var "=" expr "to" expr "do" expr ["done"] (* simple for expression *)
  | "while" expr "do" expr ["done"] (* while expression *)
  | "if" expr "then" expr {"elif" expr "then" expr} ["else" expr (* conditional
    expression *)]
  | "match" expr "with" rules (* match expression *)
  | "function" rules (* matching function expression *)
  | "let" rec functionOrValueDefns (* recursive definition *)
  | ...
```

### 7.1 For and while loops

Many programming constructs need to be repeated. The most basic example is counting, e.g., from 1 to 10 with a `for`-loop,<sup>1</sup>

```
> for i = 1 to 10 do
-   printf "%d " i
-   printfn " ";
1 2 3 4 5 6 7 8 9 10

val it : unit = ()
```

**Listing 7.1:** `fsharp`, Counting from 1 to 10 using a `for`-loop.

As this interactive script demonstrates, the identifier `i` takes all the values between 1 and 10, but in spite of its changing state, it is not mutable. Note also that the return value of the `for` expression is `()` like the `printf` functions. The `for` and `while` loops follow the syntax,

```
pat = const | ...
expr = ...
```

<sup>1</sup>Is it clear enough that the body of the loop is repeated?

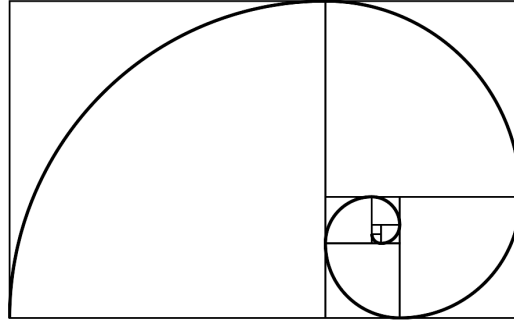


Figure 7.1: The Fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive Fibonacci numbers, and the curve in each square is the circular arc with radius of the square it is drawn in. Figure by Dicklyon <https://commons.wikimedia.org/w/index.php?curid=3730979>

```
| "for" pat "in" expr "do" expr ["done"] (* for expression *)
| "for" var "=" expr "to" expr "do" expr ["done"] (* simple for expression *)
| "while" expr "do" expr ["done"] (* while expression *)
| ...
```

Using lightweight syntax the script block between the *do* and *done* keywords may be replaced by a newline and indentation, e.g.,

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

```
1 2 3 4 5 6 7 8 9 10
```

**Listing 7.2:** countLightweight.fsx - Counting from 1 to 10 using a *for*-loop.

A more complicated example is,

Write a program that prints the  $n$ 'th Fibonacci number.

The Fibonacci numbers is the series of numbers 1, 1, 2, 3, 5, 8, 13..., where the  $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$ , and they are related to Golden spirals shown in Figure 7.1. We could solve this problem with a *for*-loop as follows,

```
let fib n =
  let mutable a = 1
  let mutable b = 1
  let mutable f = 0
  for i = 3 to n do
    f <- a + b
    a <- b
    b <- f
  f

printfn "fib(1) = 1"
printfn "fib(2) = 1"
for i = 3 to 10 do
  printfn "fib(%d) = %d" i (fib i)
```

```
fib(1) = 1
fib(2) = 1
```

```

fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55

```

Listing 7.3: fibFor.fsx - The  $n$ 'th Fibonacci number as the sum of the previous 2 numbers, which are sequentially updated from 3 to  $n$ .

The basic idea of the solution is that if we are given the  $(n - 1)$ 'th and  $(n - 2)$ 'th numbers, then the  $n$ 'th number is trivial to compute. And assume that  $\text{fib}(1)$  and  $\text{fib}(2)$  are given, then it is trivial to calculate the  $\text{fib}(3)$ . Now we have the first 3 numbers, so we disregard  $\text{fib}(1)$  and calculate  $\text{fib}(4)$  from  $\text{fib}(2)$  and  $\text{fib}(3)$ , and this process continues until we have reached the desired  $\text{fib}(n)$ . For the alternative `for`-loop, consider the problem,

Write a program that identifies prime factors of a given integer  $n$ .

Prime numbers are integers divisible only by 1 and themselves with zero remainder. Let's assume that we already have identified a list of primes from 2 to  $n$ , then we could write a program that checks the remainder as follows,

```

let primeFactorCheck n =
    printfn "%d %% i = 0?" n
    for i in [2; 3; 5; 7; 11; 13; 17] do
        printfn "i = %d? %b" i (n%i = 0)
    ()

primeFactorCheck 10

```

```

10 %% i = 0?
i = 2? true
i = 3? false
i = 5? true
i = 7? false
i = 11? false
i = 13? false
i = 17? false

```

Listing 7.4: primeCheck.fsx - Checking whether a given number has remainder zero after division by some low prime numbers.

In this example, the variable `i` runs through the elements of a list, which will be discussed in further detail in Chapter 8.

The `while`-loop is simpler than the `for`-loop and does not contain a builtin counter structure. Hence, if we are to repeat the count-to-10 program from Listing 7.1 example, it would look somewhat like,

```

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

```

```

1 2 3 4 5 6 7 8 9 10

```

**Listing 7.5:** countWhile.fsx - Count to 10 with a counter variable.

In this case, the `for`-loop is to be preferred, since more lines of code typically means more chances of making a mistake. But the `while`-loop allows for other logical structures. E.g., lets find the biggest Fibonacci number less than 100,

```
let largestFibLeq n =
    let mutable a = 1
    let mutable b = 1
    let mutable f = 0
    while f <= n do
        f <- a + b
        a <- b
        b <- f
    a

printfn "largestFibLeq(1) = 1"
printfn "largestFibLeq(2) = 1"
for i = 3 to 10 do
    printfn "largestFibLeq(%d) = %d" i (largestFibLeq i)
```

```
largestFibLeq(1) = 1
largestFibLeq(2) = 1
largestFibLeq(3) = 3
largestFibLeq(4) = 3
largestFibLeq(5) = 5
largestFibLeq(6) = 5
largestFibLeq(7) = 5
largestFibLeq(8) = 8
largestFibLeq(9) = 8
largestFibLeq(10) = 8
```

**Listing 7.6:** fibWhile.fsx - Search for the largest Fibonacci number less than a specified number.

Thus, `while`-loops are most often used, when the number of iteration cannot easily be decided, when entering the loop.

Both `for`- and `while`-loops are often associated with variables, i.e., values that change while looping. If one mistakenly used values and rebinding, then the result would in most cases be of little use, e.g.,

```
let a = 1
for i = 1 to 10 do
    let a = a + 1
    printf "(%d, %d) " a i
printf "\n"
```

```
(2, 1) (2, 2) (2, 3) (2, 4) (2, 5) (2, 6) (2, 7) (2, 8) (2, 9) (2, 10)
```

**Listing 7.7:** forScopeError.fsx - Lexical scope error. While rebinding is valid F# syntax, has little effect due to lexical scope.

I.e., the `let` expression rebinds `a` every iteration of the loop, but the value on the right-hand-side is taken lexically from above, where `a` has the value 1, so every time the result is the value 2.

## 7.2 Conditional expressions

Consider the task,

Write a function that given  $n$  writes the sentence, “I have  $n$  apple(s)”, where the plural ‘s’ is added appropriately.

For this we need to test on  $n$ ’s size, and one option is to use conditional expressions like,

```

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

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

```

**Listing 7.8:** Using conditional expression to generate different strings.

The grammar for conditional expressions is,

```

expr = ...
| "if" expr "then" expr {"elif " expr "then" expr} ["else" expr] (* conditional
  expression *)
| ...

```

where the `expr` following `if` and `elif` are *conditions*, i.e., expressions that evaluate to a boolean value. The `expr` following `then` and `else` are called *branches*, and all branches must have same type. The result of the conditional expression is the first branch, for which its condition was true. The lightweight syntax allows for the visually more simple expression of scope by use of indentation

- `if`
- `elif`
- conditions
- `then`
- `else`
- branches

```

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

```

**Listing 7.9:** Lightweight syntax allows for making blocks of code by indentation in order to make code more for easy to read.

Note that both `elif` and `else` branches are optional, which may cause problems, e.g., both `let a = if true then 3` and `let a = if true then 3 elif false then 4` will be invalid, since F# is not smart enough to realize that the type of the expression is uniquely determined. Instead F# looks for the `else` to ensure all cases have been covered, and that `a` always will be given a unique value of the same type regardless of the branches taken in the conditional statement, hence, `let a = if true then 3 else 4` is the only valid expression of the 3. However, the omitted branches are assumed to return `()`, and thus it is fine to say `let a = if true then ()` and `if true then printfn "hej"`

## 7.2.1 Programming intermezzo

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

Given an integer on decimal form, write its equivalent value on binary form

To solve this problem, consider odd numbers: They all have the property, that the least significant bit is 1, e.g.,  $1_2 = 1$ ,  $101_2 = 5$ ,  $110_2 = 6$ , and that division by 2 is equal to right-shifting by 1, e.g.,  $1_2/2 = 0.1_2 = 0.5$ ,  $101_2/2 = 10.1_2 = 2.5$ ,  $110_2/2 = 11_2 = 3$ . Thus by integer division by 2 and checking the remainder, we may sequentially read off the least significant bit. This leads to the following algorithm,

```

let dec2bin n =
  if n < 0 then
    "Illegal value"
  elif n = 0 then
    "0b0"
  else
    let mutable v = n
    let mutable str = ""
    while v > 0 do
      str <- (string (v % 2)) + str
      v <- v / 2
    "0b"+str
printfn "%d -> %s" -1 (dec2bin -1)
printfn "%d -> %s" 0 (dec2bin 0)
printfn "%d -> %s" 1 (dec2bin 1)
printfn "%d -> %s" 2 (dec2bin 2)
printfn "%d -> %s" 3 (dec2bin 3)
printfn "%d -> %s" 10 (dec2bin 10)
printfn "%d -> %s" 1023 (dec2bin 1023)

```

```

-1 -> Illegal value
0 -> 0b0
1 -> 0b1
2 -> 0b10
3 -> 0b11
10 -> 0b1010
1023 -> 0b1111111111

```

Listing 7.10: dec2bin.fsx - Using integer division and remainder to convert any positive integer to binary form.

## 7.3 Pattern matching

Conditional expressions are so common that a short-hand notation called *pattern matching* is available in F#. For the Consider the task,

· pattern  
matching

Write a function that given  $n$  writes the sentence, “I have  $n$  apple(s)”, where the plural ‘s’ is added appropriately.

For this we need to test on  $n$ ’s size, and one option is to use conditional expressions like,

```

let applesIHave n =
  match n with
  | i when i < 0 -> "I owe " + (string -i) + " apples"
  | 0 -> "I have no apples"
  | 1 -> "I have 1 apple"
  | _ -> "I have " + (string n) + " apples"

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

```

```

"I owe 3 apples"
"I owe 1 apples"

```



```
"I have no apples"
"I have 1 apple"
"I have 2 apples"
"I have 10 apples"
```

Listing 7.11: `matchWith.fsx` - Using the `match`-keyword with programming construct to vary calculation based on the input value.

Here the `match-with` keywords starts a sequence of conditions separated by the `|` lexeme, where the default operator is the `=` comparison operator, but where others can be used with the `when`. The syntax of `match` expressions is,

· `match`  
· `with`  
· `when`

```
pat = const | "_" | ...
guard = "when" expr
rule = pat [guard] -> expr
rules = "|" rule | "|" rule rules (* first '|' is optional' *)
expr = ...
  | "match" expr "with" rules (* match expression *)
  | "function" rules (* matching function expression *)
  | ...
```

As for conditional expressions, the rules are treated sequentially from first to last, and the expression following the first rule with a true condition is the result of the entire expression. The rules are versatile in their possible expression, e.g., the line `| 1 -> "I have no apples"` is equivalent to `elif n < 1 then "I have no apples"`, and the `\linline | _ -> "I have " + (string n) + " apples"`, matches the `else "I have " + (string n) + "apples"`, since the `_` lexeme is a wildcard pattern matching anything. Finally, the first rule is a guarded rule indicated by the `when` keyword, `i when i < 0 -> "I owe " + (string -i) + "apples"`. It uses the optional disregard of the `|` lexeme and is equivalent to `if n < 0 then "I owe " + (string -n) + "apples"`. Guarded rules can be any rules, and here we used the identifier `i` meaning `let i = n in if i < 0 then ...`, i.e., `n` is renamed. One way to think of guarded expressions is that `i when i < 0` is a set, and the condition is on `n` being part of the set or not.

Using lightweight syntax, the rules may be put on separate lines but must start in the column, where the `match` starts or greater.<sup>2</sup> Match with can only take one identifier, but this can be tuples for matching with combinations of identifiers, see Chapter 8 for more on tuples. A `match` expression is general but is most often seen as the initial part of a function definition. This is so common, that F# has a special syntax integrating function definitions and match with expressions using the `function` keyword,

· `function`

```
let applesIHave = function
  i when i < 0 -> "I owe " + (string -i) + " apples"
  | 0 -> "I have no apples"
  | 1 -> "I have 1 apple"
  | n -> "I have " + (string n) + " apples"

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

```
"I owe 3 apples"
"I owe 1 apples"
"I have no apples"
"I have 1 apple"
"I have 2 apples"
```

<sup>2</sup>Spec-4.0 weirdness: Offside rule for `match` is different for `function`.

```
"I have 10 apples"
```

Listing 7.12: `functionKeyword.fsx` - Function definition and `match` expressions are integrated using the `function` keyword. Compare with Listing 7.11

Comparing with Listing 7.11 notice that the function definition does not explicitly name an argument but assumes one, following the `function` follows immediately the rules, and the wildcard pattern `_` is replaced with an identifier without any guards, which thus matches everything. Replacing the wildcard pattern with a name has the advantage that this name can be used locally in the expression belonging to this rule, i.e., it acts as a `let n =` on the implicit argument of the function. Implicit arguments makes the code hard to read and, thus *the use of function definitions with the keyword `function` should be avoided*.

Advice

## 7.4 Recursive functions

Recursion is a central concept in F#. A *recursive function* is a function, which calls itself. From a compiler point of view, this is challenging, since the function is used before the compiler has completed its analysis. However, there is a technical solution for, and we will just concern ourselves with the logics of using recursion for programming. An example of a recursive function that counts from 1 to 10 similarly to Listing 7.1 is,<sup>3</sup>

· recursive  
function

```
let rec prt a b =  
    if a > b then  
        printf "\n"  
    else  
        printf "%d " a  
        prt (a + 1) b  
  
prt 1 10
```

```
1 2 3 4 5 6 7 8 9 10
```

**Listing 7.13:** `countRecursive.fsx` - Counting to 10 using recursion.

Here the `prt` calls itself repeatedly, such that the first call is `prt 1 10`, which calls `prt 2 10`, and so on until the last call `prt 10 10`. Calling `prt 11 10` would not result in recursive calls, since when `a` is higher than 10 then the *stopping criterium* is met and a newline is printed. For values of `a` smaller than or equal `b` then the recursive branch is executed. Since `prt` calls itself as the last all but the stopping condition, then this is a *tail-recursive* function. Most compilers achieve high efficiency in terms of speed and memory, so *prefer tail-recursion whenever possible*.

· stopping  
criterium  
· tail-recursive  
Advice

```
functionOrValueDefnList =  
    functionOrValueDefn  
    | functionOrValueDefn "and" functionOrValueDefnList  
expr = ...  
    | "let" rec functionOrValueDefnList (* recursive definition *)  
    | ...
```

Using recursion to calculate the Fibonacci number as Listing 7.3.

```
let rec fib n =  
    match n with  
    | i when i < 1 -> 0  
    | i when i = 1 -> 1  
    | i -> fib (n - 1) + fib (n - 2)  
  
for i = 0 to 10 do  
    printfn "fib(%d) = %d" i (fib i)
```

<sup>3</sup>A drawing showing the stack for the example would be good.

```
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
```

**Listing 7.14:** fibRecursive.fsx - The  $n$ 'th Fibonacci number using recursive.

Here we used the fact that including  $\text{fib}(0) = 0$  in the Fibonacci series also produces it using the rule  $\text{fib}(n) = \text{fib}(n-2) + \text{fib}(n-1)$ ,  $n \geq 0$ , which allowed us to define a function that is well defined for the complete set of integers. I.e., a negative argument returns 0. This is a general advice: *make functions that fails gracefully*. The recursive definition allows for recursive value definitions and defining several values and functions in one expression. Recursive values is particularly useful for defining infinite sequences, see Section 8.4.

Advice

## Chapter 8

# Tuples, Lists, Arrays, and Sequences

<sup>1</sup> F# is tuned to work with lists, and there are several built-in lists with various properties making them useful for different tasks. E.g.,

```
let solution a b c =
    let d = b ** 2.0 - 2.0 * a * c
    if d < 0.0 then
        (nan, nan)
    else
        let xp = (-b + sqrt d) / (2.0 * a)
        let xn = (-b - sqrt d) / (2.0 * a)
        (xp, xn)

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

```
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has solutions -0.7071067812 and 0.7071067812
```

**Listing 8.1:** tuplesQuadraticEq.fsx - Using tuples to gather values.

F# has 4 built-in list types: tuples, lists, arrays, and sequences following this syntax:

```
tupleList = expr | expr "," tupleList
comp-or-range-expr = comp-expr | short-comp-expr | range-expr
short-comp-expr = "for" pat "in" (expr | range-expr) "->" expr
range-expr = expr ".." expr [".." expr]
comp-expr =
    ("let" | "let!") pat "=" expr "in" comp-expr
    | ("do" | "do!") expr "in" comp-expr
    | ("use" | "use!") pat = expr "in" comp-expr
    | ("yield" | "yield!") expr
    | ("return" | "return!") expr
    | "if" expr "then" comp-expr ["else" comp-expr]
    | "match" expr "with" comp-rules
    | "try" comp-expr "with" comp-rules
    | "try" comp-expr "finally" expr
    | "while" expr "do" expr ["done"]
    | "for" ident "=" expr "to" expr "do" comp-expr ["done"]
    | "for" pat "in" expr-or-range-expr "do" comp-expr ["done"]
    | comp-expr ";" comp-expr
```

---

<sup>1</sup>possibly add maps and sets as well.

```

| expr
comp-rule = pat pattern-guardopt ">-" comp-expr
comp-rules = comp-rule | comp-rule '|' comp-rules
expr = ...
| tupleList
| "[" comp-or-range-expr "]" (* computed list expression *)
| "[" comp-or-range-expr "]" (* computed array expression *)
| expr "{" comp-or-range-expr "}" (* computation expression *)
| ...

```

<sup>2</sup>Tuples are a direct extension of constants. They are immutable and do not have concatenations nor indexing operations. This is in contrast to lists. Lists are also immutable, but have a simple syntax for concatenation and indexing. Arrays are mutable lists, and support higher order structures such as tables and 3 dimensional arrays. Sequences are like lists, but with the added advantage of a very flexible construction mechanism, and the option of representing infinite long sequences. In the following, we will present these data structures in detail.

## 8.1 Tuples

*Tuples* are unions of immutable types,

· tuple

```

tupleList = expr | expr "," tupleList
expr = ...
| tupleList
| ...

```

and they are identified by the `,` lexeme. Most often the tuple is enclosed in parentheses, but that is not required. Consider the triplet, also known as a 3-tuple, `(2, true, "hello")` in interactive mode,

```

> let tp = (2, true, "hello")
- printfn "%A" tp;;
(2, true, "hello")

val tp : int * bool * string = (2, true, "hello")
val it : unit = ()

```

**Listing 8.2:** fsharpi, Definition of a tuple.

The values `2`, `true`, and `"hello"` are *members*, and the number of elements of a tuple is its *length*. From the response of F# we see that the tuple is inferred to have the type `int * bool * string`, where the `*` is cartesian product between the three sets. Notice, that tuples can be products of any types and have lexical scope like value and function bindings. Notice also, that a tuple may be printed as a single entity by the `%A` placeholder. In the example, we bound `tp` to the tuple, the opposite is also possible,

· member  
· length

```

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

val deconstructNPrint : 'a * 'b * 'c -> unit
val it : unit = ()

```

**Listing 8.3:** fsharpi, Definition of a tuple.

<sup>2</sup>Spec-4.0: grammar for list and array expressions are subsets of computed list and array expressions.

In this a function is defined that takes 1 argument, a 3-tuple, and which is bound to a tuple with 3 named members. Since we used the %A placeholder in the `printfn` function, then the function is generic and can be called with 3-tuples of different types. Note, *don't confuse a function of n arguments with a function of an n-tuple*. The later has only 1 argument, and the difference is the , 's. Another example is `let solution a b c = ...`, which is the beginning of the function definition in Listing 8.1. It is a function of 3 arguments, while `let solution (a, b, c) = ...` would be a function of 1 argument, which is a 3-tuple. Functions of several arguments makes currying easy, i.e., we could define a new function which fixes the quadratic term to be 0 as `let solutionToLinear = solution 0.0`, that is, without needing to specify anything else. With tuples, we would need the slightly more complicated, `let solutionToLinear (b, c) = solution (0.0, b, c)`.

Advice

Tuples comparison are defined similarly as strings. Tuples of different lengths are different. For tuples of equal length, then they are compared element by element. E.g., `(1,2) = (1,3)` is false, while `(1,2) = (1,2)` is true. The `<>` operator is the boolean negation of the `=` operator. For the `<`, `<=`, `>`, and `>=` operators, the strings are ordered alphabetically like, such that `('a', 'b', 'c') < ('a', 'b', 's')` and `('a', 'b', 's') < ('c', 'o', 's')` is true, that is, the `<` operator on two tuples is true, if the left operand should come before the right, when sorting alphabetically like.

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

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

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

```
('a', 'b', 'c') < ('d', 'e', 'f') is true
('a', 'b', 'c') < ('a', 'b', 'b') is false
('a', 'b', 'c') < ('a', 'b', 'd') is true
```

**Listing 8.4:** tupleCompare.fsx - Tuples are compared as strings are compared alphabetically.

The algorithm for deciding the boolean value of `(a1, a2) < (b1, b2)` is as follows: we start by examining the first elements, and if `!a1` and `b1` are different, then the `(a1, a2) < (b1, b2)` is equal to `a1 < b1`. If `!a1` and `b1` are equal, then we move onto the next letter and repeat the investigation. The `<=`, `>`, and `>=` operators are defined similarly.

Binding tuples to mutables does not make the tuple mutable, e.g.,

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

```
1, 2, (1, 2)
3, 2, (1, 2)
```

**Listing 8.5:** tupleOfMutables.fsx - A mutable change value, but the tuple defined by it does not refer to the new value.

However, tuples may be mutable such that new tuple values can be assigned to it, e.g., in the Fibonacci example, we can write a more compact script by using mutable tuples and the `fst` and `snd` functions as follows.

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

for i = 0 to 10 do
  printfn "fib(%d) = %d" i (fib i)
```

```
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
```

**Listing 8.6:** fibTuple.fsx - Calculating Fibonacci numbers using mutable tuple.

In this example, the central computation has been packed into a single line, `prev <- (snd prev, (fst prev) + (snd prev))`, where both the calculation of  $\text{fib}(n) = \text{fib}(n-2) + \text{fib}(n-1)$  and the rearrangement of memory to hold the new values  $\text{fib}(n)$  and  $\text{fib}(n-1)$  based on the old values  $\text{fib}(n-2) + \text{fib}(n-1)$ . While this may look elegant and short there is the risk of *obfuscation*, i.e., writing compact code that is difficult to read, and in this case, an unprepared reader of the code may not easily understand the computation nor appreciate its elegance without an accompanying explanation. Hence, *always keep an eye out for compact and concise ways to write code, but never at the expense of readability.*

· obfuscation

Advice

## 8.2 Lists

*Lists* are unions of immutable values of the same type and have a more flexible structure than tuples. Its grammar follows *computational expressions*, which is very rich and shared with arrays and sequences, and we will delay a discussion on most computational expressions to Section 8.4, and here just consider a subset of the grammar:

· list  
· computational expressions

```
comp-or-range-expr = comp-expr | .. | range-expr
range-expr = expr ".." expr [".." expr]
comp-expr =
  ...
  | comp-expr ";" comp-expr
  | expr
expr = ...
  | "[" comp-or-range-expr "]" (* computed list expression *)
  | ...
```

Simple examples of a list grammars are, `[expr; expr; ... ; expr]`, `[expr ".."expr]`, `[expr ".."expr ".."expr]`, e.g., an explicit list `let lst = [1; 2; 3; 4; 5]`, which may be

written shortly as *range expression* as `let lst = [1 .. 5]`, and ranges may include a step size `let lst = [1 .. 2 .. 5]`, which is the same as `let lst = [1; 3; 5]`.  
 Lists may be indexed and concatenated much like strings, e.g.,

· range  
 expression

```
let printList (lst : int list) =
    for elm in lst do
        printf "%A " elm
    printfn ""

let printListAlt (lst : int list) =
    for i = 0 to lst.Length - 1 do
        printf "%A " lst.[i]
    printfn ""

let a = [1; 2;]
let b = [3; 4; 5]
let c = a @ b
let d = 0 :: c
printfn "%A, %A, %A, %A" a b c d
printList d
printListAlt d
```

```
[1; 2], [3; 4; 5], [1; 2; 3; 4; 5], [0; 1; 2; 3; 4; 5]
0 1 2 3 4 5
0 1 2 3 4 5
```

**Listing 8.7:** listIndexing.fsx - Examples of list concatenation, indexing.

A list type is identified with the `list` keyword, as here a list of integers is `int list`. Above, we used the `@` and `::` concatenation operators, the `.[]` index method, and the `Length` property. Notice, as strings, list elements are counted from 0, and thus the last element has `lst.Length - 1`. In `printList` the `for-in` is used, which runs loops through each element of the list and assigns it to the identifier `elm`. This is in contrast to `printListAlt`, which uses the `for-to` keyword and explicitly represents the index `i`. Explicit representation of the index makes more complicated programs, and thus increases the chances of programming errors. Hence, *for-in is to be preferred over for-to*. Lists support slicing identically to strings, e.g.,

· @  
 · ::  
 · .[]  
 · Length

Advice

```
let lst = ['a' .. 'g']
printfn "%A" lst.[0]
printfn "%A" lst.[3]
printfn "%A" lst.[3..]
printfn "%A" lst[..3]
printfn "%A" lst.[1..3]
printfn "%A" lst.[*]
```

```
'a'
'd'
['d'; 'e'; 'f'; 'g']
['a'; 'b'; 'c'; 'd']
['b'; 'c'; 'd']
['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g']
```

**Listing 8.8:** listSlicing.fsx - Examples of list slicing. Compare with Listing 5.27.

Lists are well suited for recursive functions and pattern matching with, e.g., `match-with` as illustrated in the next example:

```
let rec printListRec (lst : int list) =
    match lst with
```



```

elm::rest ->
    printf "%A " elm
    printListRec rest
| _ ->
    printfn ""

let a = [1; 2; 3; 4; 5]
printListRec a

```

```
1 2 3 4 5
```

**Listing 8.9:** listPatternMatching.fsx - Examples of list concatenation, indexing.

The pattern `1::rest` is the pattern for the first element followed by a list of the rest of the list. This pattern matches all lists except an empty list, hence `rest` may be empty. Thus the wildcard pattern matching anything including the empty list, will be used only when `lst` is empty.

*Pattern matching* with lists is quite powerful, consider the following problem:

· pattern  
matching

Given a list of pairs of course names and course grades, calculate the average grade.

A list of course names and grades is `[("name1", grade1); ("name2", grade2); ...]`. Let's take a recursive solution. First problem will be to iterate through the list. For this we can use pattern matching similarly to Listing 8.9 with `(name, grade)::rest`. The second problem will be to calculate the average. The average grade is the sum all grades and divide by the number of grades. Assume that we already have made a function, which calculates the `sum` and `n`, the sum and number of elements, for `rest`, then all we need is to add `grade` to the `sum` and 1 to `n`. For an empty list, `sum` and `n` should be 0. Thus we arrive at the following solution,

```

let averageGrade courseGrades =
    let rec sumNCount lst =
        match lst with
        | (title, grade)::rest ->
            let (sum, n) = sumNCount rest
            (sum + grade, n + 1)
        | _ -> (0, 0)

    let (sum, n) = sumNCount courseGrades
    (float sum) / (float n)

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

printfn "Course and grades:\n%A" courseGrades
printfn "Average grade: %.1f" (averageGrade courseGrades)

```

```

Course and grades:
[("Introduction to programming", 95); ("Linear algebra", 80);
 ("User Interaction", 85)]
Average grade: 86.7

```

**Listing 8.10:** avgGradesRec.fsx - Calculating a list of average grades using recursion and pattern matching.

Pattern matching and appending is a useful combination, if we wish to produce new from old lists. E.g., a function returning a list of squared entries of its argument can be programmed as,

```
let rec square a =
```

```

match a with
  elm :: rest -> elm*elm :: (square rest)
  | _ -> []

let a = [1 .. 10]
printfn "%A" (square a)

```

```
[1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```

**Listing 8.11:** listSquare.fsx - Using pattern matching and list appending elements to lists.

This is a prototypical functional programming style solution, and which uses the `::` for 2 different purposes: First the list `[1 .. 10]` is first matched with `1 :: [2 .. 10]`, and then we assume that we have solved the problem for `square rest`, such that all we need to do is append `1*1` to the beginning output from `square rest`. Hence we get, `square [1 .. 10]  $\hookrightarrow$  1 * 1 :: square [2 .. 10]  $\hookrightarrow$  1 * 1 :: (2 * 2 :: square [3 .. 10])  $\hookrightarrow$  ... 1 * 1 :: (2 * 2 :: ... 10 * 10 :: [])`, where the stopping criterium is reached, when the `elm :: rest` does not match with `a`, hence it is empty, which does match the wildcard pattern `_`. More on functional programming in Section 16

The basic properties and members of lists are summarized in Table 8.1. In addition, Lists have many other built-in functions, which will be discussed in Chapter 13 and Part III.

It is possible to make multidimensional lists as lists of lists, e.g.,

```

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

```

```
[1; 2]
2
2
```

**Listing 8.12:** listMultidimensional.fsx - A ragged multidimensional list, built as lists of lists, and its indexing.

The example shows a *ragged multidimensional list*, since each row has different number of elements. The indexing of a particular element is not elegant, which is why arrays are often preferred in F#.

· ragged multidimensional list

## 8.3 Arrays

### 8.3.1 1 dimensional arrays

1 dimensional arrays or just arrays for short are mutable lists of the same type and follow a similar syntax as lists. Its grammar follows *computational expressions*, which will be discussed in Section 8.4. Here we consider a subset of the grammar:

· computational expressions

```

comp-or-range-expr = comp-expr | ... | range-expr
comp-expr =
  | comp-expr ";" comp-expr
  | expr
expr = ...
  | "[" comp-or-range-expr "]" (* computed array expression *)
  | ...

```

Thus the creation of arrays is identical to lists, but there is no explicit operator support for appending and concatenation, e.g.,

```

let printArray (arr : int array) =
  for elm in arr do

```

| Function name | Example  | Description   |
|---------------|--|---|
| Length        | <pre>&gt; [1; 2; 3].Length;; val it : int = 3 &gt; let a = [1; 2; 3] in a.Length;; val it : int = 3</pre>  | The number of elements in a list                          |
| List.Empty    | <pre>&gt; let a : int list = List.Empty;;  val a : int list = []  &gt; let b = List&lt;int&gt;.Empty;;  val b : int list = []</pre>                                    | An empty list of specified type                           |
| IsEmpty       | <pre>&gt; [1; 2; 3].IsEmpty;; val it : bool = false &gt; let a = [1; 2; 3] in a.IsEmpty;; val it : bool = false</pre>  | Compare with the empty list                               |
| Item          | <pre>&gt; [1; 2; 3].Item 1;; val it : int = 2 &gt; let a = [1; 2; 3] in a.Item 1;; val it : int = 2</pre>  | Indexing  |
| Head          | <pre>&gt; [1; 2; 3].Head;; val it : int = 1 &gt; let a = [1; 2; 3] in a.Head;; val it : int = 1</pre>  | The first element in the list.<br>Exception if empty.     |
| Tail          | <pre>&gt; [1; 2; 3].Tail;; val it : int list = [2; 3] &gt; let a = [1; 2; 3] in a.Tail;; val it : int list = [2; 3]</pre>  | The list except its first element.<br>Exception if empty. |
| Cons          | <pre>&gt; list.Cons (1, [2; 3]);; val it : int list = [1; 2; 3] &gt; 1 :: [2; 3];; val it : int list = [1; 2; 3]</pre>   | Append an element to the front of the list                |
| @             | <pre>&gt; [1] @ [2; 3];; val it : int list = [1; 2; 3] &gt; [1; 2] @ [3; 4];; val it : int list = [1; 2; 3; 4] &gt; [1; 2] @ [3];; val it : int list = [1; 2; 3]</pre> | Concatenate two lists                                     |

Table 8.1: Basic properties and members of lists.

```

    printf "%d " elm
    printf "\n"

let printArrayAlt (arr : int array) =
    for i = 0 to arr.Length - 1 do
        printf "%A " arr.[i]
    printfn ""

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

```

```

[|1; 2|], [|3; 4; 5|], [|1; 2; 3; 4; 5|]
1 2 3 4 5
1 2 3 4 5

```

**Listing 8.13:** arrayCreation.fsx - Creating arrays with a syntax similarly to lists.

The array type is defined using the `array` keyword or alternatively the `[]` lexeme. Arrays cannot be resized, but are mutable,

```

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]

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

printArray A
square A
printArray A

```

```

1 2 3 4 5
1 4 9 16 25

```

**Listing 8.14:** arrayReassign.fsx - Arrays are mutable in spite the missing `mutable` keyword.

Notice that in spite the missing `mutable` keyword, the function `square` still had the *side-effect* of squaring all entries in `A`. Arrays only support direct pattern matching, e.g.,

```

let name2String (arr : string array) =
    match arr with
    | [| first; last|] -> last + ", " + first
    | _ -> ""

let listNames (arr : string array array) =
    let mutable str = ""
    for a in arr do
        str <- str + name2String a + "\n"
    str

let A = [| [| "Jon"; "Sporring"|]; [| "Alonzo"; "Church"|]; [| "John"; "McCarthy"|] |]
printf "%s" (listNames A)

```

```
Sporring, Jon
Church, Alonzo
McCarthy, John
```

**Listing 8.15:** `arrayPatternMatching.fsx` - Only simple pattern matching is allowed for arrays.

The given example is the first example of a 2-dimensional array, which can be implemented as arrays of arrays and here written as `string array array`. In Section ?? will 2 and higher dimensional arrays be discussed. 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 arr = [|'a' .. 'g'|]
printfn "%A" arr.[0]
printfn "%A" arr.[3]
printfn "%A" arr.[3..]
printfn "%A" arr[..3]
printfn "%A" arr.[1..3]
printfn "%A" arr.[*]
```

```
'a'
'd'
[|'d'; 'e'; 'f'; 'g'|]
[|'a'; 'b'; 'c'; 'd'|]
[|'b'; 'c'; 'd'|]
[|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'|]
```

Listing 8.16: `arraySlicing.fsx` - Examples of array slicing. Compare with Listing 8.8 and Listing 5.27.

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 many which will be discussed in Chapter ??.

### 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,

```
let arr = [| [|1|]; [|1; 2|]; [|1; 2; 3|] |]

for row in arr do
  for elm in row do
    printf "%A " elm
  printf "\n"
```

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

Listing 8.17: `arrayJagged.fsx` - An array of arrays. When row lengths are of non-equal elements, then it is a Jagged array.

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

```
let pownArray (arr : int array array) p =
  for i = 1 to arr.Length - 1 do
```

```

    for j = 1 to arr.[i].Length - 1 do
        arr.[i].[j] <- pown arr.[i].[j] p

let printArrayOfArrays (arr : int array array) =
    for row in arr do
        for elm in row do
            printf "%3d " elm
        printf "\n"

let A = [| [| 1 .. 4 |]; [| 1 .. 2 .. 7 |]; [| 1 .. 3 .. 10 |] |]
pownArray A 2
printArrayOfArrays A

```

```

1    2    3    4
1    9   25   49
1   16   49  100

```

Listing 8.18: arrayJaggedSquare.fsx - A rectangular array. Notice, the `for-in` cannot be used in `pownArray`

, e.g., `for row in arr do for elm in row do elm <- pown elm p done done` since the iterator value `\lstinline!elm!` is not mutable even though `arr` is an array. In fact, square arrays of dimensions 2 to 4 are so common that F# has built-in modules for their support. In the following 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 arr = Array2D.create 3 4 0
for i = 0 to (Array2D.length1 arr) - 1 do
    for j = 0 to (Array2D.length2 arr) - 1 do
        arr.[i,j] <- j * Array2D.length1 arr + i
printfn "%A" arr

```

```

[[0; 3; 6; 9]
 [1; 4; 7; 10]
 [2; 5; 8; 11]]

```

**Listing 8.19:** array2D.fsx - Creating a 3 by 4 rectangular arrays of integers.

Notice that the indexing uses a slightly different notation `'[,]'` and the length functions are also slightly different. The statement `A.Length` would return the total number of elements in the array, in this case 12. As can be seen, the `printf` supports direct printing of the 2 dimensional array. Higher dimensional arrays support slicing, e.g.,

```

let arr = Array2D.create 3 4 0
for i = 0 to (Array2D.length1 arr) - 1 do
    for j = 0 to (Array2D.length2 arr) - 1 do
        arr.[i,j] <- j * Array2D.length1 arr + i
printfn "%A" arr.[2,3]
printfn "%A" arr.[1..,3..]
printfn "%A" arr[..1,*]
printfn "%A" arr.[1,*]
printfn "%A" arr.[1..1,*]

```

```

11
[[10]
 [11]]
[[0; 3; 6; 9]
 [1; 4; 7; 10]]
[[1; 4; 7; 10]]

```

```
[[1; 4; 7; 10]]
```

**Listing 8.20:** array2DSlicing.fsx - Examples of Array2D slicing. Compare with Listing 8.19.

Note that in almost all cases, slicing produces a sub rectangular 2 dimensional array except for `arr.[1,*]`, which is an array, as can be seen by the single `.`. In contrast, `A.[1..1,*]` is an `Array2D`. Note also, that `printfn` typesets 2 dimensional arrays as `[[ ... ]]` and not `[|[ ... |]]`, which can cause confusion with lists of lists.<sup>3</sup>

`Array2D` and higher have a number of built-in functions that will be discussed in Chapter ??.

## 8.4 Sequences

A computational expression is a small program, whose result is list, for example, `let lst = [for i = 1 to 5 do yield i done]`

```
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 8.21:** arrayJaggedCompExpr.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 8.22:** arrayJaggedSquareCompExpr.fsx -

<sup>3</sup>`Array2D.ToString` produces `[[ ... ]]` and not `[|[ ... |]]`, which can cause confusion.

In fact, square arrays of dimensions 2 to 4 are so common that F# 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 8.23:** array2DCompExpr.fsx -

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.



## 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.

Remember

- exn type Spec-4.0 Chapter 18.1
- Spec-4.0 Section 18.2.8

...

## Chapter 10

# Testing programs

...

# Chapter 11

## Input/Output

<sup>1</sup>

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
```

---

<sup>1</sup>Spec-4.0 Section 18.2.9

```

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 System.String

The list of built-in methods accessible with the dot notation is defined in `System.String` class and is long. Here follows short descriptions of some useful methods:

`Compare(String, String)` Compares two specified String objects and returns an integer that indicates their relative position in the sort order.

`CompareOrdinal(String, String)` Compares two specified String objects by evaluating the numeric values of the corresponding Char objects in each string.

`CompareOrdinal(String, Int32, String, Int32, Int32)` Compares substrings of two specified String objects by evaluating the numeric values of the corresponding Char objects in each substring.

`CompareTo(Object)` Compares this instance with a specified Object and indicates whether this instance precedes, follows, or appears in the same position in the sort order as the specified Object.

`CompareTo(String)` Compares this instance with a specified String object and indicates whether this instance precedes, follows, or appears in the same position in the sort order as the specified String.

`Concat(Object)` Creates the string representation of a specified object.

`Concat(Object[])` Concatenates the string representations of the elements in a specified Object array.

`Concat(IEnumerable(String))` Concatenates the members of a constructed `IEnumerable(T)` collection of type String.

`Concat(String[])` Concatenates the elements of a specified String array.

`Concat(Object, Object)` Concatenates the string representations of two specified objects.

`Concat(String, String)` Concatenates two specified instances of String.

`Concat(Object, Object, Object)` Concatenates the string representations of three specified objects.

`Concat(String, String, String)` Concatenates three specified instances of String.

`Concat(Object, Object, Object, Object)` Concatenates the string representations of four specified objects and any objects specified in an optional variable length parameter list.

`Concat(String, String, String, String)` Concatenates four specified instances of String.

`Concat(T)(IEnumerable(T))` Concatenates the members of an `IEnumerable(T)` implementation.

**Contains** Returns a value indicating whether the specified String object occurs within this string.

**Copy** Creates a new instance of String with the same value as a specified String.

**CopyTo** Copies a specified number of characters from a specified position in this instance to a specified position in an array of Unicode characters.

**EndsWith(String)** Determines whether the end of this string instance matches the specified string.

**EndsWith(String, StringComparison)** Determines whether the end of this string instance matches the specified string when compared using the specified comparison option.

**EndsWith(String, Boolean, CultureInfo)** Determines whether the end of this string instance matches the specified string when compared using the specified culture.

**Equals(Object)** Determines whether this instance and a specified object, which must also be a String object, have the same value. (Overrides Object.Equals(Object).)

**Equals(String)** Determines whether this instance and another specified String object have the same value.

**Equals(String, String)** Determines whether two specified String objects have the same value.

**Equals(String, StringComparison)** Determines whether this string and a specified String object have the same value. A parameter specifies the culture, case, and sort rules used in the comparison.

**Equals(String, String, StringComparison)** Determines whether two specified String objects have the same value. A parameter specifies the culture, case, and sort rules used in the comparison.

**Finalize** Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection. (Inherited from Object.)

**Format(String, Object)** Replaces one or more format items in a specified string with the string representation of a specified object.

**Format(String, Object[])** Replaces the format item in a specified string with the string representation of a corresponding object in a specified array.

**Format(IFormatProvider, String, Object[])** Replaces the format item in a specified string with the string representation of a corresponding object in a specified array. A specified parameter supplies culture-specific formatting information.

**Format(String, Object, Object)** Replaces the format items in a specified string with the string representation of two specified objects.

**Format(String, Object, Object, Object)** Replaces the format items in a specified string with the string representation of three specified objects.

**GetEnumerator** Retrieves an object that can iterate through the individual characters in this string.

**GetHashCode** Returns the hash code for this string. (Overrides Object.GetHashCode().)

**GetType** Gets the Type of the current instance. (Inherited from Object.)

**GetTypeCode** Returns the TypeCode for class String.

**IndexOf(Char)** Reports the zero-based index of the first occurrence of the specified Unicode character in this string.

**IndexOf(String)** Reports the zero-based index of the first occurrence of the specified string in this instance.



**IndexOf(Char, Int32)** Reports the zero-based index of the first occurrence of the specified Unicode character in this string. The search starts at a specified character position.

**IndexOf(String, Int32)** Reports the zero-based index of the first occurrence of the specified string in this instance. The search starts at a specified character position.

**IndexOf(String, StringComparison)** Reports the zero-based index of the first occurrence of the specified string in the current String object. A parameter specifies the type of search to use for the specified string.

**IndexOf(Char, Int32, Int32)** Reports the zero-based index of the first occurrence of the specified character in this instance. The search starts at a specified character position and examines a specified number of character positions.

**IndexOf(String, Int32, Int32)** Reports the zero-based index of the first occurrence of the specified string in this instance. The search starts at a specified character position and examines a specified number of character positions.

**IndexOf(String, Int32, StringComparison)** Reports the zero-based index of the first occurrence of the specified string in the current String object. Parameters specify the starting search position in the current string and the type of search to use for the specified string.

**IndexOf(String, Int32, Int32, StringComparison)** Reports the zero-based index of the first occurrence of the specified string in the current String object. Parameters specify the starting search position in the current string, the number of characters in the current string to search, and the type of search to use for the specified string.

**IndexOfAny(Char[])** Reports the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters.

**IndexOfAny(Char[], Int32)** Reports the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters. The search starts at a specified character position.

**IndexOfAny(Char[], Int32, Int32)** Reports the zero-based index of the first occurrence in this instance of any character in a specified array of Unicode characters. The search starts at a specified character position and examines a specified number of character positions.

**Insert** Returns a new string in which a specified string is inserted at a specified index position in this instance.

**Intern** Retrieves the system's reference to the specified String.

**IsInterned** Retrieves a reference to a specified String.

**IsNormalized()** Indicates whether this string is in Unicode normalization form C.

**IsNormalized(NormalizationForm)** Indicates whether this string is in the specified Unicode normalization form.

**IsNullOrEmpty** Indicates whether the specified string is a null reference (Nothing in Visual Basic) or an Empty string.

**IsNullOrWhiteSpace** Indicates whether a specified string is a null reference (Nothing in Visual Basic), empty, or consists only of whitespace characters.

**Join(String, IEnumerable(String))** Concatenates the members of a constructed IEnumerable(T) collection of type String, using the specified separator between each member.

**Join(String, Object[])** Concatenates the elements of an object array, using the specified separator between each element.

**Join(String, String[])** Concatenates all the elements of a string array, using the specified separator between each element.

**Join(String, String[], Int32, Int32)** Concatenates the specified elements of a string array, using the specified separator between each element.

**Join(T)(String, IEnumerable(T))** Concatenates the members of a collection, using the specified separator between each member.

**LastIndexOf(Char)** Reports the zero-based index position of the last occurrence of a specified Unicode character within this instance.

**LastIndexOf(String)** Reports the zero-based index position of the last occurrence of a specified string within this instance.

**LastIndexOf(Char, Int32)** Reports the zero-based index position of the last occurrence of a specified Unicode character within this instance. The search starts at a specified character position.

**LastIndexOf(String, Int32)** Reports the zero-based index position of the last occurrence of a specified string within this instance. The search starts at a specified character position.

**LastIndexOf(String, StringComparison)** Reports the zero-based index of the last occurrence of a specified string within the current String object. A parameter specifies the type of search to use for the specified string.

**LastIndexOf(Char, Int32, Int32)** Reports the zero-based index position of the last occurrence of the specified Unicode character in a substring within this instance. The search starts at a specified character position and examines a specified number of character positions.

**LastIndexOf(String, Int32, Int32)** Reports the zero-based index position of the last occurrence of a specified string within this instance. The search starts at a specified character position and examines a specified number of character positions.

**LastIndexOf(String, Int32, StringComparison)** Reports the zero-based index of the last occurrence of a specified string within the current String object. Parameters specify the starting search position in the current string, and type of search to use for the specified string.

**LastIndexOf(String, Int32, Int32, StringComparison)** Reports the zero-based index position of the last occurrence of a specified string within this instance. Parameters specify the starting search position in the current string, the number of characters in the current string to search, and the type of search to use for the specified string.

**LastIndexOfAny(Char[])** Reports the zero-based index position of the last occurrence in this instance of one or more characters specified in a Unicode array.

**LastIndexOfAny(Char[], Int32)** Reports the zero-based index position of the last occurrence in this instance of one or more characters specified in a Unicode array. The search starts at a specified character position.

**LastIndexOfAny(Char[], Int32, Int32)** Reports the zero-based index position of the last occurrence in this instance of one or more characters specified in a Unicode array. The search starts at a specified character position and examines a specified number of character positions.

**MemberwiseClone** Creates a shallow copy of the current Object. (Inherited from Object.)

**Normalize()** Returns a new string whose textual value is the same as this string, but whose binary representation is in Unicode normalization form C.

**Normalize(NormalizationForm)** Returns a new string whose textual value is the same as this string, but whose binary representation is in the specified Unicode normalization form.

**PadLeft(Int32)** Returns a new string that right-aligns the characters in this instance by padding them with spaces on the left, for a specified total length.

**PadLeft(Int32, Char)** Returns a new string that right-aligns the characters in this instance by padding them on the left with a specified Unicode character, for a specified total length.

**PadRight(Int32)** Returns a new string that left-aligns the characters in this string by padding them with spaces on the right, for a specified total length.

**PadRight(Int32, Char)** Returns a new string that left-aligns the characters in this string by padding them on the right with a specified Unicode character, for a specified total length.

**Remove(Int32)** Returns a new string in which all the characters in the current instance, beginning at a specified position and continuing through the last position, have been deleted.

**Remove(Int32, Int32)** Returns a new string in which a specified number of characters in this instance beginning at a specified position have been deleted.

**Replace(Char, Char)** Returns a new string in which all occurrences of a specified Unicode character in this instance are replaced with another specified Unicode character.

**Replace(String, String)** Returns a new string in which all occurrences of a specified string in the current instance are replaced with another specified string.

**Split(Char[])** Returns a string array that contains the substrings in this instance that are delimited by elements of a specified Unicode character array.

**Split(Char[], Int32)** Returns a string array that contains the substrings in this instance that are delimited by elements of a specified Unicode character array. A parameter specifies the maximum number of substrings to return.

**Split(Char[], StringSplitOptions)** Returns a string array that contains the substrings in this string that are delimited by elements of a specified Unicode character array. A parameter specifies whether to return empty array elements.

**Split(String[], StringSplitOptions)** Returns a string array that contains the substrings in this string that are delimited by elements of a specified string array. A parameter specifies whether to return empty array elements.

**Split(Char[], Int32, StringSplitOptions)** Returns a string array that contains the substrings in this string that are delimited by elements of a specified Unicode character array. Parameters specify the maximum number of substrings to return and whether to return empty array elements.

**Split(String[], Int32, StringSplitOptions)** Returns a string array that contains the substrings in this string that are delimited by elements of a specified string array. Parameters specify the maximum number of substrings to return and whether to return empty array elements.

**StartsWith(String)** Determines whether the beginning of this string instance matches the specified string.

**StartsWith(String, StringComparison)** Determines whether the beginning of this string instance matches the specified string when compared using the specified comparison option.

**StartsWith(String, Boolean, CultureInfo)** Determines whether the beginning of this string instance matches the specified string when compared using the specified culture.

**Substring(Int32)** Retrieves a substring from this instance. The substring starts at a specified character position.

**Substring(Int32, Int32)** Retrieves a substring from this instance. The substring starts at a specified character position and has a specified length.

**ToCharArray()** Copies the characters in this instance to a Unicode character array.

**ToCharArray(Int32, Int32)** Copies the characters in a specified substring in this instance to a Unicode character array.

**ToLower()** Returns a copy of this string converted to lowercase.

**ToLower(CultureInfo)** Returns a copy of this string converted to lowercase, using the casing rules of the specified culture.

**ToLowerInvariant** Returns a copy of this String object converted to lowercase using the casing rules of the invariant culture.

**ToString()** Returns this instance of String; no actual conversion is performed. (Overrides Object.ToString().)

**ToString(IFormatProvider)** Returns this instance of String; no actual conversion is performed.

**ToUpper()** Returns a copy of this string converted to uppercase.

**ToUpper(CultureInfo)** Returns a copy of this string converted to uppercase, using the casing rules of the specified culture.

**ToUpperInvariant** Returns a copy of this String object converted to uppercase using the casing rules of the invariant culture.

**Trim()** Removes all leading and trailing whitespace characters from the current String object.

**Trim(Char[])** Removes all leading and trailing occurrences of a set of characters specified in an array from the current String object.

**TrimEnd** Removes all trailing occurrences of a set of characters specified in an array from the current String object.

**TrimStart** Removes all leading occurrences of a set of characters specified in an array from the current String object.

## 13.2 List, arrays, and sequences

In Table 13.1. 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
    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 13.1:** `arrayReassignModule.fsx` -

|            |   |
|------------|---|
| 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.   |

Table 13.1: Some built-in procedures in the Array module for arrays (from <https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/fsharp-core-library-reference>)

|            |  |
|------------|--|
| 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 13.2: Some built-in procedures in the Array2D module for arrays (from <https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/fsharp-core-library-reference>)

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 13.2:** `flowForListsIndexModule.fsx` -

Both cases avoid the use of variables and side-effects which is a big advantage for code safety. There are a bit few built-in procedures for 2 dimensional array types, some of which are summarized in Table 13.2

## 13.3 Mutable Collections

`System.Collections.Generic`

### 13.3.1 Mutable lists

`List`, `LinkedList`

### 13.3.2 Stacks

`Stack`

### 13.3.3 Queues

`Queue`

### 13.3.4 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
"ymle m )i lsauy cat.a o) eNtalmtucalu. haet giitCile ifeI mgrE"euubehce tl
is datcira rattTueep ac(i( Ft ldfu0fCl]na o il elatx.atr)p nl=earu1 a.
rcVySce elpidihhn.an t.ymr.p.maz ahdil cT Aeu r:r."
```

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 hist2DToCumulativeProbability 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 = hist2DToCumulativeProbability 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

## Types and measures

### 15.1 Unit of Measure

F# allows for assigning *unit of measure* to the following types,

· unit of measure

sbyte, int, int16, int32, int64, single, float32, float, and decimal.

by using the syntax,

```
"[<Measure>] type" unit-name [ "=" unit-expr ]
```

and then use "<unit-name >" as suffix for literals. E.g., defining unit of measure 'm' and 's', then we can make calculations like,

```
> [<Measure>] type m
- [<Measure>] type s
- let a = 3<m/s^2>
- let b = a * 10<s>
- let c = 4 * b;;

[<Measure>]
type m
[<Measure>]
type s
val a : int<m/s ^ 2> = 3
val b : int<m/s> = 30
val c : int<m/s> = 120
```

**Listing 15.1:** fsharp, floating point and integer numbers may be assigned unit of measures.

However, if we mixup unit of measures under addition, then we get an error,

```
> [<Measure>] type m
- [<Measure>] type s
- let a = 1<m>
- let b = 1<s>
- let c = a + b;;

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

/Users/sporring/repositories/fsharpNotes/stdin(63,13): error FS0001: The unit
of measure 's' does not match the unit of measure 'm'
```

Listing 15.2: fsharp, unit of measures adds an extra layer of types for syntax checking at compile time.

Unit of measures allow for  $*$ ,  $/$ , and  $\sim^1$  for multiplication, division and exponentiation. Values with units can be casted to *unit-less* values by casting, and back again by multiplication as,

· unit-less

```
> [<Measure>] type m
- let a = 2<m>
- let b = int a
- let c = b * 1<m>;;

[<Measure>]
type m
val a : int<m> = 2
val b : int = 2
val c : int<m> = 2
```

**Listing 15.3:** fsharp, type casting unit of measures.

Compound symbols can be declared as,

```
> [<Measure>] type s
- [<Measure>] type m
- [<Measure>] type kg
- [<Measure>] type N = kg * m / s^2;;

[<Measure>]
type s
[<Measure>]
type m
[<Measure>]
type kg
[<Measure>]
type N = kg m/s ^ 2
```

**Listing 15.4:** fsharp, aggregated unit of measures.

For fans of the metric system there is the International System of Units, and these are built-in in `Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols` and give in Table 15.1. Hence, using the predefined unit of seconds, we may write,

```
> let a = 10.0<Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols.s>;;

val a : float<Data.UnitSystems.SI.UnitSymbols.s> = 10.0
```

**Listing 15.5:** fsharp, SI unit of measures are built-in.

To make the use of these predefined symbols easier, we can import them into the present scope by the *open* keyword,

· open

```
> open Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols;;
> let a = 10.0<s>;;

val a : float<s> = 10.0
```

**Listing 15.6:** fsharp, simpler syntax by importing, but beware of namespace pollution.

The *open* keyword should be used with care, since now all the bindings in `Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols` have been imported into the present scope, and since we most likely do not know, which bindings have been used by the programmers of `Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols`, we do not know which identifiers to avoid, when using *let* statements. We have obtained, what is known as *namespace pollution*. Read more about namespaces in Part IV. Using unit of measures is advisable for calculations involving real-world values, since some semantical errors of arithmetic expressions may be discovered by checking the resulting unit of measure.

· namespace  
pollution

<sup>1</sup>Spec-4.0: this notation is inconsistent with **\*\*** for float exponentiation.

| Unit | Description   |
|------|---|
| A    | Ampere, unit of electric current.                                 |
| Bq   | Becquerel, unit of radioactivity.                                 |
| C    | Coulomb, unit of electric charge, amount of electricity.          |
| cd   | Candela, unit of luminous intensity.                              |
| F    | Farad, unit of capacitance.                                       |
| Gy   | Gray, unit of an absorbed dose of radiation.                      |
| H    | Henry, unit of inductance.  |
| Hz   | Hertz, unit of frequency.   |
| J    | Joule, unit of energy, work, amount of heat.                      |
| K    | Kelvin, unit of thermodynamic (absolute) temperature.             |
| kat  | Katal, unit of catalytic activity.                                |
| kg   | Kilogram, unit of mass.   |
| lm   | Lumen, unit of luminous flux.                                     |
| lx   | Lux, unit of illuminance.   |
| m    | Metre, unit of length.  |
| mol  | Mole, unit of an amount of a substance.                           |
| N    | Newton, unit of force.  |
| ohm  | Unitnames.o SI unit of electric resistance.                       |
| Pa   | Pascal, unit of pressure, stress.                                 |
| s    | Second, unit of time.   |
| S    | Siemens, unit of electric conductance.                            |
| Sv   | Sievert, unit of dose equivalent.                                 |
| T    | Tesla, unit of magnetic flux density.                             |
| V    | Volt, unit of electric potential difference, electromotive force. |
| W    | Watt, unit of power, radiant flux.                                |
| Wb   | Weber, unit of magnetic flux.                                     |

Table 15.1: International System of Units.



## Chapter 16

# Functional programming

## Part IV

# Structured programming

# Chapter 17

## Namespaces and Modules

Things to remember:

- difference between .fs and .fsx Spec-4.0 Chapter 12.1 and 12.3
- signature files and their usefulness

A script file consists of a sequence of *module elements*

· module  
elements

```
script-file = implementation-file

implementation-file =
  namespace-decl-groupList
  | named-module
  | anonymous-module

namespace-decl-groupList = namespace-decl-group | namespace-decl-group namespace-decl-groupList

named-module = "module" long-ident module-elems

anonymous-module = module-elems

module-elems = module-elem | module-elem module-elems

namespace-decl-group = "namespace" long-ident module-elems | "namespace" global module-elems

module-elem =
  module-function-or-value-defn type-defns
  | exception-defn
  | module-defn
  | module-abbrev
  | import-decl compiler-directive-decl
```

F# source code units are made up of declarations grouped using namespaces, type definitions, and module definitions. A file may contain multiple namespaces each defining types and modules, these in turn may contain function and value definitions, which in turn contains expressions.<sup>1</sup>

With no leading namespace or module declaration, then F# will immediately insert a module, where the name of the module is the same as the file name with capitalized first letter.<sup>2</sup>

---

<sup>1</sup>Spec-4.0 Chapter 10.

<sup>2</sup>[https://en.wikibooks.org/wiki/F\\_Sharp\\_Programming/Modules\\_and\\_Namespaces](https://en.wikibooks.org/wiki/F_Sharp_Programming/Modules_and_Namespaces)

Namespaces is an optional hierarchial categorization of modules, classes, and other namespaces primarily used to avoid naming conflicts. There is no default namespace, and namespaces may contain type definitions but not function and value definitions. Namespace do not work in script-fragments.<sup>3</sup>

---

<sup>3</sup><https://fsharpforfunandprofit.com/posts/organizing-functions/>

## Chapter 18

# Object-oriented programming

Things to remember:

- upcast and downcast `upcast`, `:`, `downcast`, `?:`
- boxing `(box 5) :?> int;;`, see Spec-4.0 chapter 18.2.6.
- obj type Spec-4.0 chapter 18.1
- boxing Spec-4.0 Section 18.2.6

# 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:

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

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,

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

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. The left-most bit is called the *most significant bit*, and the right-most bit is called the *least significant bit*. Due to typical organization of computer memory, 8 binary digits is called a *byte*, and 32 digits a *word*.

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 integers 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,

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

· decimal number  
· base  
· decimal point  
· digit

· bit  
· binary

· most significant  
bit  
· least significant  
bit  
· byte  
· word  
· octal  
· hexadecimal

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

| Dec | Bin   | Oct | Hex | Dec | Bin    | Oct | Hex |
|-----|-------|-----|-----|-----|--------|-----|-----|
| 0   | 0     | 0   | 0   | 32  | 100000 | 40  | 20  |
| 1   | 1     | 1   | 1   | 33  | 100001 | 41  | 21  |
| 2   | 10    | 2   | 2   | 34  | 100010 | 42  | 22  |
| 3   | 11    | 3   | 3   | 35  | 100011 | 43  | 23  |
| 4   | 100   | 4   | 4   | 36  | 100100 | 44  | 24  |
| 5   | 101   | 5   | 5   | 37  | 100101 | 45  | 25  |
| 6   | 110   | 6   | 6   | 38  | 100110 | 46  | 26  |
| 7   | 111   | 7   | 7   | 39  | 100111 | 47  | 27  |
| 8   | 1000  | 10  | 8   | 40  | 101000 | 50  | 28  |
| 9   | 1001  | 11  | 9   | 41  | 101001 | 51  | 29  |
| 10  | 1010  | 12  | a   | 42  | 101010 | 52  | 2a  |
| 11  | 1011  | 13  | b   | 43  | 101011 | 53  | 2b  |
| 12  | 1100  | 14  | c   | 44  | 101100 | 54  | 2c  |
| 13  | 1101  | 15  | d   | 45  | 101101 | 55  | 2d  |
| 14  | 1110  | 16  | e   | 46  | 101110 | 56  | 2e  |
| 15  | 1111  | 17  | f   | 47  | 101111 | 57  | 2f  |
| 16  | 10000 | 20  | 10  | 48  | 110000 | 60  | 30  |
| 17  | 10001 | 21  | 11  | 49  | 110001 | 61  | 31  |
| 18  | 10010 | 22  | 12  | 50  | 110010 | 62  | 32  |
| 19  | 10011 | 23  | 13  | 51  | 110011 | 63  | 33  |
| 20  | 10100 | 24  | 14  | 52  | 110100 | 64  | 34  |
| 21  | 10101 | 25  | 15  | 53  | 110101 | 65  | 35  |
| 22  | 10110 | 26  | 16  | 54  | 110110 | 66  | 36  |
| 23  | 10111 | 27  | 17  | 55  | 110111 | 67  | 37  |
| 24  | 11000 | 30  | 18  | 56  | 111000 | 70  | 38  |
| 25  | 11001 | 31  | 19  | 57  | 111001 | 71  | 39  |
| 26  | 11010 | 32  | 1a  | 58  | 111010 | 72  | 3a  |
| 27  | 11011 | 33  | 1b  | 59  | 111011 | 73  | 3b  |
| 28  | 11100 | 34  | 1c  | 60  | 111100 | 74  | 3c  |
| 29  | 11101 | 35  | 1d  | 61  | 111101 | 75  | 3d  |
| 30  | 11110 | 36  | 1e  | 62  | 111110 | 76  | 3e  |
| 31  | 11111 | 37  | 1f  | 63  | 111111 | 77  | 3f  |

Table 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

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

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

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

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

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](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. Each code-point has a number of attributes such as the *unicode general category*. Presently more than 128,000 code points covering 135 modern and historic writing systems, and obtained at <http://www.unicode.org/Public/UNIDATA/UnicodeData.txt>, which includes the code point, name, and general category.

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.

The general category is used in grammars to specify valid characters, e.g., in naming identifiers in F#. Some categories and their letters in the first 256 code points are shown in Table B.5.

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

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

- UTF-8
- UTF-16

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

Table 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.

| General category | Code points   | Name  |
|------------------|---|---|
| Lu               | U+0041–U+005A, U+00C0–U+00D6, U+00D8–U+00DE         | Upper case letters                          |
| Ll               | U+0061–U+007A, U+00B5, U+00DF–U+00F6, U+00F8–U+00FF | Lower case letter                           |
| Lt               | None  | Digraphic letter, with first part uppercase |
| Lm               | None  | Modifier letter                             |
| Lo               | U+00AA, U+00BA                                      | Gender ordinal indicator                    |
| Nl               | None  | Letterlike numeric character                |
| Pc               | U+005F  | Low line                                    |
| Mn               | None  | Nonspacing combining mark                   |
| Mc               | None  | Spacing combining mark                      |
| Cf               | U+00AD  | Soft Hyphen                                 |

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

## 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.,

```
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.

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

'{ ... }' 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.

## Appendix D

# Language Details

<sup>1</sup>

---

<sup>1</sup>Somewhere I should possibly talk about Lightweight Syntax, Spec-4.0 Chapter 15.1

| Operator   | Associativity | Description                      |
|--|---------------|----------------------------------|
| ident "<"types ">"   | Left          | High-precedence type application |
| ident "("expr ")"  | Left          | High-precedence application      |
| "."  | Left          |                                  |
| prefixOp   | Left          | All prefix operators             |
| " rule   | Left          | Pattern matching rule            |
| ident expr,<br>"lazy" expr,<br>"assert" epxr                                     | Left          |                                  |
| "**"opChar   | Right         | Exponent like                    |
| "*"opChar, "/"opChar,<br>"%"opChar   | Left          | Infix multiplication like        |
| "-"opChar, "+"opChar   | Left          | Infix addition like              |
| ":?"'  | None          |                                  |
| "::"'  | Right         |                                  |
| "^" opChar   | Right         |                                  |
| "!="opChar, "<"opChar,<br>">"opChar, "=",<br>" "opChar, "&"opChar,<br>"\$"opChar | Left          | Infix addition like              |
| ":>", ":?>"  | Right         |                                  |
| "&", "&&"  | Left          | Boolean and like                 |
| "or", "  "   | Left          | Boolean or like                  |
| ", "   | None          |                                  |
| ":="   | Right         |                                  |
| "->"   | Right         |                                  |
| "if"   | None          |                                  |
| "function", "fun",<br>"match", "try"   | None          |                                  |
| "let"  | None          |                                  |
| "; "   | Right         |                                  |
| "  "   | Left          |                                  |
| "when"   | Right         |                                  |
| "as"   | Right         |                                  |

Table D.1: Precedence and associativity of operators. Operators in the same row has same precedence. See Listing 6.28 for the definition of `prefixOp`

# 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

. [], 28  
abs, 20  
acos, 20  
asin, 20  
atan2, 20  
atan, 20  
bignum, 17  
byte[], 17  
byte, 17  
ceil, 20  
char, 14  
cosh, 20  
cos, 20  
decimal, 17  
double, 17  
eprintfn, 41  
eprintf, 41  
exn, 14  
exp, 20  
failwithf, 41  
float32, 17  
float, 14  
floor, 20  
fprintfn, 41  
fprintf, 41  
ignore, 41  
int16, 17  
int32, 17  
int64, 17  
int8, 17  
int, 14  
it, 14  
log10, 20  
log, 20  
max, 20  
min, 20  
nativeint, 17  
obj, 14  
pown, 20  
printfn, 41  
printf, 40, 41  
round, 20  
sbyte, 17  
sign, 20  
single, 17

sinh, 20  
sin, 20  
sprintf, 41  
sqrt, 20  
stderr, 41  
stdout, 41  
string, 14  
tanh, 20  
tan, 20  
uint16, 17  
uint32, 17  
uint64, 17  
uint8, 17  
unativeint, 17  
unit, 14

American Standard Code for Information Inter-  
change, 106

and, 24  
anonymous function, 37  
ASCII, 106  
ASCIIbetical order, 27, 106

base, 14, 102  
Basic Latin block, 107  
Basic Multilingual plane, 107  
basic types, 14  
binary, 102  
binary number, 16  
binary operator, 20  
binary64, 102  
binding, 10  
bit, 16, 102  
block, 34  
blocks, 107  
boolean and, 23  
boolean or, 23  
branches, 54  
byte, 102

character, 16  
class, 19, 28  
code point, 16, 107  
compiled, 8  
computational expressions, 62, 65  
conditions, 54

- Cons, 65
- console, 8
- currying, 38
- debugging, 9
- decimal number, 14, 102
- decimal point, 14, 102
- Declarative programming, 5
- digit, 14, 102
- dot notation, 28
- double, 102
- downcasting, 19
- EBNF, 14, 110
- encapsulate code, 35
- encapsulation, 38, 43
- exception, 26
- exclusive or, 26
- executable file, 8
- expression, 10, 19
- expressions, 6
- Extended Backus-Naur Form, 14, 110
- Extensible Markup Language, 46
- floating point number, 14
- format string, 10
- fractional part, 14, 19
- function, 12
- Functional programming, 6, 87
- functions, 6
- generic function, 36
- Head, 65
- hexadecimal, 102
- hexadecimal number, 16
- HTML, 48
- Hyper Text Markup Language, 48
- IEEE 754 double precision floating-point format, 102
- Imperativ programming, 87
- Imperative programming, 5
- implementation file, 8
- infix notation, 23
- infix operator, 19
- integer division, 25
- integer number, 14
- interactive, 8
- IsEmpty, 65
- Item, 65
- jagged arrays, 68
- keyword, 10
- Latin-1 Supplement block, 107
- Latin1, 106
- least significant bit, 102
- Length, 65
- length, 60
- lexeme, 12
- lexical scope, 12, 36
- lexically, 32
- lightweight syntax, 30, 32
- list, 62
- List.Empty, 65
- literal, 14
- literal type, 17
- machine code, 87
- member, 19, 60
- method, 28
- module elements, 98
- modules, 8
- most significant bit, 102
- Mutable data, 42
- namespace, 19
- namespace pollution, 94
- NaN, 104
- nested scope, 12, 34
- newline, 17
- not, 24
- not a number, 104
- obfuscation, 62
- object, 28
- Object oriented programming, 87
- Object-oriented programming, 6
- objects, 6
- octal, 102
- octal number, 16
- operand, 35
- operands, 20
- operator, 20, 23, 35
- or, 24
- overflow, 25
- overshadow, 12
- overshadows, 34
- pattern matching, 55, 64
- precedence, 23
- prefix operator, 20
- Procedural programming, 87
- procedure, 38
- production rules, 110
- ragged multidimensional list, 65
- range expression, 63
- reals, 102

- recursive function, 57
- reference cells, 44
- remainder, 25
- rounding, 19
- run-time error, 26
  
- scientific notation, 16
- scope, 12, 33
- script file, 8
- script-fragments, 8
- side-effect, 67
- side-effects, 38, 44
- signature file, 8
- slicing, 68
- state, 5
- statement, 10
- statements, 5, 87
- states, 87
- stopping criterium, 57
- string, 10, 16
- Structured programming, 6
- subnormals, 104
  
- Tail, 65
- tail-recursive, 57
- terminal symbols, 110
- truth table, 24
- tuple, 60
- type, 10, 14
- type casting, 18
- type declaration, 10
- type inference, 9, 10
- type safety, 36
  
- unary operator, 20
- underflow, 25
- Unicode, 16
- unicode general category, 107
- Unicode Standard, 107
- unit of measure, 93
- unit-less, 94
- unit-testing, 9
- upcasting, 19
- UTF-16, 107
- UTF-8, 107
  
- variable, 42
- verbatim, 18
  
- whitespace, 17
- whole part, 14, 19
- word, 102
  
- XML, 46
- xor, 26