Learning to program with F#

Jon Sporring

September 11, 2016

Contents

Chapter 1

Preface

This book has been written as an introduction to programming for novice programmers. It is used on the first programming course at the University of Copenhagen's bachelor in computer science program. It has been typeset in LaTeX, and all programs have been developped and tested in Mono version 4.4.1.

Jon Sporring
Associate Professor, Ph.d.
Department of Computer Science,
University of Copenhagen
September 11, 2016

Chapter 2

Introduction

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

2.1 How to learn to program

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

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

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

3. Practice: If you want to be a good programmer, then there is only one way: practice, practice, practice! It has been estimated that to master anything, then you have to have spent at least 10000 hours of practice, so get started logging hours! It of course matters, what you practice. This book teaches 3 different programming themes. The point is that programming is thinking, and the scaffold that you use, shapes your thoughts. It is therefore important to recognize this scaffold, and to have the ability to choose that which suits your ideas and your goals best. And

the best way to expand your abilities is to both sharpen your present abilities, push yourself into new territory, and trying something new. Do not be afraid to make errors or be frustrated at first. These are the experiences that make you grow.

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

2.2 How to solve problems

Programming is the act of solving a problem by writing a program to be executed on a computer. A general method for solving problems was given by George Pólya [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 description of the problem. Is all information for finding the solution available or is something missing?

Design a plan: Good designs mean that programs are faster to program easier to 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 the 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 Pólya's list is a useful guide, but not a failsafe approach. Always approach problem solving with an open mind.

2.3 Approaches to programming

This book focuses on 3 fundamentally different approaches to programming:

Imperative programming, which is a type of programming where *statements* are used 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 recipe 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, which emphasises *what a program shall accomplish* but not *how*. We will consider Functional programming as an example of declarative programming language. A *functional programming* language 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].

· Declarative programming
· Functional programming
· functional programming
· functions
· expressions

Structured programming, which emphasises organisation of code in units with well defined interfaces and isolation of internal states and code from other parts of the program. We will focus on Object-oriented programming as the example of structured programming. *Object-orientered 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.

· Structured programming
· Object-orientered programming
· objects

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.

2.4 Why use F#

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

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

Strongly typed F# is strongly typed, reducing the number of run-time errors: F# is picky, and will not allow the programmer to mix up types such as integers and strings. This is a great advantage for large programs.

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

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

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

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

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

2.5 How to read this book

Learning to program requires mastering a programming language, however most programming languages contains details that are rarely used or used in contexts far from a specific programming topic. Hence, this book 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, meaning that it has strong support for functional programming, but F# also supports imperative and object oriented programming. It also has strong support for parallel programming and information rich programs. It was originally developed for Microsoft's .Net platform, but is available as open source for many operating systems through Mono. In this text we consider F# 4.0 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 `fsharpc` and execution of the compiled code is performed using the `mono` command. The various forms of fsharp programs are identified by suffixes:

· interactive
· compiled
· console

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

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

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

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

`.exe` An *executable file*, e.g., `gettingStartedStump.exe`

· implementation file
· signature file
· script file
· executable file

The implementation, signature, and script files are all typically compiled to produce an executable file, but syntactical 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*. Modules are collections of smaller programs used by other programs, which will be discussed in detail in Part IV.

· 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:

Listing 3.1:

```
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. The following dialogue demonstrates the workflow, where what the user types has been highlighted by a box:

Listing 3.2: An interactive session.

```
$ 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,

Listing 3.3: Using the interpreter to execute a script.

```
$ fsharpi gettingStartedStump.fsx
3
```

Finally, compiling and executing the code is performed as,

Listing 3.4: Compiling and executing a script.

```
$ 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`". In contrast, compiled code does not need to be recompiled to be run again, only re-executed using `"$ mono gettingStartedStump.exe"`.On a Macbook Pro, with a 2.9 Ghz Intel Core i5, the time the various stages takes for this script are:

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

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

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

· 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,

Problem 4.1:

What is the sum of 357 and 864?

we have written the following program in F#,

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

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

```
1221
```

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

To solve the problem, 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

· statement
· `let`
· keyword
· binding
· expression
· format string
· string

must match the value *type*, that is, here `c` is of type integer, whereas the format string `%A` matches many types. · type

Types are a central concept in F#. In the script 4.1 we bound values of integer type to names. There are several different integer types in F#, here we used the one called `int`. The values were not *declared* to have these types, instead the types were *inferred* by F#. Had we typed these statements line by line in an interactive session, then we would have seen the inferred types: · type declaration · type inference

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

```
> let a = 357;;

val a : int = 357

> let b = 864;;

val b : int = 864

> let c = a + b;;

val c : int = 1221

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

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

Were we to solve a slightly different problem,

Problem 4.2:

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,

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

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

```
1221.0
```

On the surface, this could appear as an almost negligible change, but the set of integers and the set of real numbers (floats) require quite different representations, in order to be effective on a computer, and as a consequence, the implementation of their operations such as addition are very different. Thus, although the response is an integer, it has type `float`, which is indicated by `1221.0` 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,

Listing 4.4, typeInferenceError.fsx:
Mixing types is often not allowed.

```
> let a = 357;;

val a : int = 357

> let b = 863.4;;

val b : float = 863.4

> let c = a + b;;

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

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

we see that binding a name to a number without a decimal point is inferred to be integer, while when binding to a number with a decimal point, then the type is inferred to be a float, and when trying to add values of integer and floating point, 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., [1]

· lexical scope

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

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

```



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

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

[1] Todo: **When command is omitted, then error messages have unwanted blank lines.**

Listing 4.6, blocksNNames.fsx:
Names may be reused when separated by the lexeme ;;.

```
> let a = 357;;

val a : int = 357

> let a = 864;;

val a : int = 864
```

then rebinding did not cause an error. The difference is that the *;; lexeme*, which specifies the end of a *script-fragment*. A lexeme is a letter or a word, which the F# considers as an atomic unit. Script-fragments may be defined both in scripts and in interactive mode, and rebinding is not allowed at the outermost level in script-fragments.

· ;;
· lexeme
· script-fragment

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

· function

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

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

```
1221
```

Entering the function into an interactive session will illustrate the inferred type, the function `sum` has: `val sum : x:int -> y:int -> int`. The `->` is the mapping operator in the sense that functions are mappings between sets. The type of the function `sum`, should be read as `val sum : x:int -> (y: int -> int)`, that is, `sum` takes an integer and returns a function, which takes an integer and returns an integer. 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,

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

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

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

> let c = sum 357.6 863.4;;

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

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

A remedy is to define the function in the same script-fragment as it is used, i.e,

Listing 4.9, typesNBlockInference.fsx:
Defining a function together with its use, makes F# infer the appropriate types.

```
> 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
```

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 the number 3, and if we type the number `3` in an interactive session at the input prompt, then F# responds as follows,

· type
· literal

Listing 5.1, firstType.fsx:
Typing the number 3.

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

What this means is that F# has inferred the type to be ***`int`*** and bound it to the identifier ***`it`***. 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.,

· `int`
· `it`

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

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

Each literal 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, ***`bool`*** for boolean values, ***`char`*** for characters, and ***`string`*** for strings of characters are the most common types of literals. A table of all *basic types* predefined in F# is given in Table 5.1. Besides these built-in types, F# is designed such that it is easy to define new types.

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

Humans like to use the *decimal number* system for representing numbers. Decimal numbers are *base* 10,

· decimal number
· base

Metatype	Type name	Description
Boolean	**bool**	Boolean values true or false
Integer	**int**	Integer values from -2,147,483,648 to 2,147,483,647
	byte	Integer values from 0 to 255
	sbyte	Integer values from -128 to 127
	int32	Synonymous with int
	uint32	Integer values from 0 to 4,294,967,295
Real	**float**	64-bit IEEE 754 floating point value from $-\infty$ to ∞
	double	Synonymous with float
Character	**char**	Unicode character
	string	Unicode sequence of characters
None	**unit**	No value denoted
Object	**obj**	An object
Exception	**exn**	An exception

Table 5.1: List of some of the 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 20, and for exceptions see Chapter 11.

which that a value is represented as two sequences of decimal digits separated by a *decimal point*, where each *digit* can have values $d \in \{0, 1, 2, \ldots, 9\}$, 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*. As an example `35.7` is a decimal number, whose value is $3 \cdot 10^1 + 5 \cdot 10^0 + 7 \cdot 10^{-1}$, and `128` is an integer, whose value is $1 \cdot 10^2 + 2 \cdot 10^1 + 8 \cdot 10^0$. In F# a decimal number is called a *floating point number* and in this text we use *Extended Backus-Naur Form* (*EBNF*) to describe the grammar of F#. In EBNF, the grammar describing a decimal number is,

· decimal point
· digit
· whole part
· fractional part
· integer
· floating point number
· Extended Backus-Naur Form
· EBNF

Listing 5.3: Decimal numbers.

```
dDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
dInt = dDigit {dDigit}; (*no spaces*)
dFloat = dInt "." {dDigit}; (*no spaces*)
```

In EBNF `dDigit`, `dInt`, and `dFloat` are names of tokens, while `"0"`, `"1"`, ..., `"9"`, and `"."` are terminals. Tokens and terminals together with formatting rules describe possible sequences, which are valid. E.g., a `dDigit` is defined by the = notation to be either `0` or `1` or ...or `9`, as signified by the `|` syntax. The definition of a token is ended by a `;`. The `"{ }"` in EBNF signfies zero or more repetitions of its content, such that a `dInt` is, e.g., `dDigit`, `dDigit dDigit`, `dDigit dDigit dDigit dDigit` and so on. Since a `dDigit` is any decimal digit, we conclude that `3`, `45`, and `0124972930485738` are examples of `dInt`. A `dFloat` is the concatenation of one or more digits, a dot, and zero or more digits, such as `0.4235`, `3.`, but not `.5` nor `.`. Sometimes EBNF implicitly allows for spaces between tokens and terminals, so here we have used the comments notation `(* *)` to explicitly remind ourselves, that no spaces are allowed between the whole part, decimal point, and the fractional part. A complete description of EBNF is given in Appendix C.

Floating point numbers may alternatively be given using *scientific notation*, such as `3.5e-4` and `4e2`, where the `e`-notation is translated to a value as `3.5e-4` $= 3.5 \cdot 10^{-4} = 0.00035$, and `4e2` $= 4 \cdot 10^2 = 400$. To describe this in EBNF we write

· scientific notation

Listing 5.4: Scientific notation.

```
sFloat = (dInt | dFloat) ("e" | "E" ) ["+" | "-"] dInt; (*no spaces*)
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. Internally, programs and data is all represented as bits, hence F# has a strong support for binary numbers. A *binary number* consists of a sequence of binary digits separated by a decimal point, where each digit can have values $b \in \{0, 1\}$, and the base is 2. E.g., the binary number $101.01_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 5.25$. Binary numbers are closely related to *octal* and *hexadecimal numbers*, where octals uses 8 as basis, and where each octal digit can be represented by exactly 3 bits, while hexadecimal numbers uses 16 as basis, and where each hexadecimal digit can be written in binary using exactly 4 bits. The hexadecimal digits uses `0`–`9` to represent the values 0–9 and `a`–`f` in lower or alternatively upper case to represent the values 10-15. Octals and hexadecimals 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

Listing 5.5: Binary, hexadecimal, and octal numbers.

```
bDigit = "0" | "1";
oDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7";
xDigit =
  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
  | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" | "f";
bitInt = "0" ("b" | "B") bDigit {bDigit}; (*no spaces*)
octInt = "0" ("o" | "O") oDigit {oDigit}; (*no spaces*)
hexInt = "0" ("x" | "X") xDigit {xDigit}; (*no spaces*)
xInt = bitInt | octInt | hexInt;
int = dInt | xInt;
```

For example the value 367 in base 10 may be written as a `dInt` integer as `367`, as a `bitInt` binary number as `0b101101111`, as a `octInt` octal number as `0o557`, and as a `hexInt` hexadecimal number as `0x16f`. 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. The EBNF for characters is,

· character
· Unicode
· code point

Listing 5.6: Character escape sequences.

```
codePoint = ?Any unicode codepoint?;
escapeChar =
  "\" ("b" | "n" | "r" | "t" | "\" | '"' | "'" | "a" | "f" | "v")
  | "\u" xDigit xDigit xDigit xDigit
  | "\U" xDigit xDigit xDigit xDigit xDigit xDigit xDigit xDigit
  | "\" dDigit dDigit dDigit;  (*no spaces*)
char = "'" codePoint | escapeChar "'"; (*no spaces*)
```

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`, `\UXXXXXXXX` allow for the full specification of any code point. Examples of a `char` are `'a'`, `'_'`, `'\n'`, and `'\065'`.

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.

A *string* is a sequence of characters enclosed in double quotation marks, · string

Listing 5.7: Strings.

```
stringChar = char - '"';
string = '"' { stringChar }  '"';
verbatimString = '@"' {char - ('"' | '\"' )| '""'} '"';
```

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

Listing 5.8: Whitespace and newline.

```
whitespace = " " {" "};
newline = "\n" | "\r" "\n";
```

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

Listing 5.9, stringLiterals.fsx:
Examples of string literals.

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

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 og suffix as shown in the Table 5.3. Examples are, · literal type

<table>
<tr><th>type</th><th>EBNF</th><th>Examples</th></tr>
<tr><td><code>int</code>, <code>int32</code></td><td><code>(dInt | xInt)["l"]</code></td><td><code>3</code></td></tr>
<tr><td><code>uint32</code></td><td><code>(dInt | xInt)("u"| "ul")</code></td><td><code>3u</code></td></tr>
<tr><td><code>byte</code>, <code>uint8</code></td><td><code>((dInt | xInt)"uy")| (char "B")</code></td><td><code>3uy</code></td></tr>
<tr><td><code>byte[]</code></td><td><code>["@"] string "B"</code></td><td><code>"abc"B</code> and <code>"@http:\\"B</code></td></tr>
<tr><td><code>sbyte</code>, <code>int8</code></td><td><code>(dInt | xInt)"y"</code></td><td><code>3y</code></td></tr>
<tr><td><code>float</code>, <code>double</code></td><td><code>float | (xInt "LF")</code></td><td><code>3.0</code></td></tr>
<tr><td><code>string</code></td><td><code>simpleString |</code></td><td><code>"a \"quote\".\n"</code></td></tr>
<tr><td></td><td><code>'@"'{(char − ('"'| '\"'))| '""'} '"'|</code></td><td><code>@"a ""quote"".\n"</code></td></tr>
</table>

Table 5.3: List of literal type. No spacing is allowed between the literal and the prefix or suffix.

Listing 5.10, namedLiterals.fsx:
Named and implied literals.

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

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

Listing 5.11, stringVerbatim.fsx:
Examples of a string literal.

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

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

Listing 5.12, upcasting.fsx:
Casting an integer to a floating point number.

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

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

Listing 5.13, castingBooleans.fsx:
Casting booleans.

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

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

· member
· class
· namespace

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

Listing 5.14, downcasting.fsx:
Fractional part is removed by downcasting.

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

Here we typecasted to a lesser type, in the sense that the set of integers is a subset of floating point numbers, and this is called *downcasting*. The opposite is called *upcasting* and is often non-destructive, as Listing 5.12 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

Listing 5.15, rounding.fsx:
Fractional part is removed by downcasting.

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

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 5.15 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. In the example, `+` is the operator, and it is an infix operator, since it takes values on its left and right side. The grammar for expressions are defined recursively, and some of it is given by,

· expression
· infix operator

Listing 5.16: Expressions.

```
const = byte | sbyte | int32 | uint32 | int | ieee64 | char | string
  | verbatimString | "false" | "true" | "()";
sliceRange =
  expr
  | expr ".." (*no space between expr and ".."*)
  | ".." expr (*no space between expr and ".."*)
  | expr ".." expr (*no space between expr and ".."*)
  | "*";
expr = ...
  | const (*a const value*)
  | "(" expr ")" (*block*)
  | expr expr (*application*)
  | expr infixOp expr (*infix application*)
  | prefixOp expr (*prefix application*)
  | expr ".[" expr "]" (*index lookup, no space before "."*)
  | expr ".[" sliceRange "]" (*index lookup, no space before "."*)
```

Recursion means that a rule or a function is used by the rule or function itself in its definition, e.g., in the definition of `expression`, the token `expression` occurs both on the left and the right side of the = symbol. 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 postive 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 such as addition, subtraction, multiplication, division, and exponentiation using the `+`, `-`, `*`, `/`, `**` binary operators respectively. Not all operators are defined for all types, e.g., addition is defined for integer and float types as well as for characters and strings, but multiplication is only defined for integer and floating point types. A complete list of built-in operators on basic types is shown in Table E.1 and E.2 and a range of mathematical functions shown in Table E.3.

· operator
· operands
· binary operator
· prefix operator
· unary operator

The concept of *precedence* is an important concept in arithmetic expressions.[1] If parentheses are omitted in Listing 5.15, then F# will interpret the expression as `(int 357.6) + 0.5`, which is erroneous, since addition of an integer with a float is undefined. This is an example of precedence, i.e., function evaluation takes precedence over addition meaning that it is performed before addition. Consider the arithmetic expression,

· precedence

Listing 5.17, simpleArithmetic.fsx:
A simple arithmetic expression.

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

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

· infix notation
· operator
· precedence

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

a	b	a && b	a || b	not a
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Table 5.4: Truth table for boolean 'and', 'or', and 'not' operators. Value 0 is false and 1 is true.

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

Listing 5.18, precedence.fsx:
Precedences rules define implicite parentheses.

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

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

5.3 Boolean arithmetic

Boolean arithmetic is the basis of almost all computers and particularly important for controlling program flow, which will be discussed in Chapter 8. Boolean values are one of 2 possible values, true or false, which is also sometimes written as 1 and 0. Basic operations on boolean values are '*and*', '*or*', and '*not*', which in F# is written as the binary operators `&&`, `||`, and the function `not`. Since the domain of boolean values is so small, then all possible combination of input on these values can be written on tabular form, known as a *truth table*, and the truth tables for the basic boolean operators and functions is shown in Table 5.4. A good mnemonics for remembering the result of the 'and' and 'or' operators is to use 1 for true, 0 for false, multiplication for the boolean 'and' operator, and addition for boolean 'or' operator, e.g., true and false in this mnemonic translates to $1 \cdot 0 = 0$, and the results translates back to the boolean value false. In F# the truth table for the basic boolean operators is reproduced by,

· and
· or
· not
· truth table

Listing 5.19, truthTable.fsx:
Boolean operators and truth tables.

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

val it : unit = ()
```

In Listing 5.19 we used the `printfn` function, to present the results of many expressions on something that resembles a tabular form. The spacing produced using the `printfn` function is not elegant, and in Section 6.4 we will discuss better options for producing more beautiful output. Notice, that the arguments for `printfn` was given on the next line with indentation. The indentation is an important part of telling F#, which part of what you write belongs together. This is an example of the so-called lightweight syntax. Generally, F# ignores newlines and whitespaces except when using the ligthweight syntax, and the examples of the difference between regular and lightweight syntax is discussed in Chapter 6.

5.4 Integer arithmetic

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

Table E.1, E.2, and E.3 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 may cause *overflow* and *underflow*. E.g., the range of the integer type `sbyte` is $[-128 \ldots 127]$, which causes problems in the following example,

· overflow
· underflow

Listing 5.20, overflow.fsx:
Adding integers may cause overflow.

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

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

Listing 5.21, underflow.fsx:
Subtracting integers may cause underflow.

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

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

The overflow error in Listing 5.20 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`,

Listing 5.22, overflowBits.fsx:
The left most bit is interpreted differently for signed and unsigned integers, which gives rise to potential overflow errors.

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

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

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

· integer division
· remainder

Listing 5.23, integerDivisionRemainder.fsx:
Integer division and remainder operators.

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

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

Listing 5.24, integerDivisionRemainderLossless.fsx:
Integer division and remainder is a lossless representation of an integer, compare with Listing 5.23.

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

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

Notice that neither overflow nor underflow error gave rise to an error message, which is why such bugs are difficult to find. Dividing any non-zero number with 0 is infinite, which is also outside the domain

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

Table 5.5: Boolean exclusive or truth table.

of any of the integer types, but in this case, F# casts an *exception*, · exception

Listing 5.25, integerDivisionByZeroError.fsx:
Integer division by zero causes an exception run-time error.

```
> 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 + 0
      x000a1> in <filename unknown>:0
Stopped due to error
```

The output looks daunting at first sight, but the first and last line of the error message are the most important parts, which tells us what exception was cast and why the program stopped. The middle are technical details concerning which part of the program caused this, and can be ignored for the time being. Exceptions are a type of *run-time error*, and are treated in Chapter 11 · run-time error

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

Listing 5.26, integerPown.fsx:
Integer exponent function.

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

which is equal to 2^5.

For binary arithmetic on integers, the following operators are available: `leftOp <<< rightOp`, which shifts the bit pattern of `leftOp` `rightOp` positions to the left while inserting 0's to right; `leftOp >>> rightOp`, which shifts the bit pattern of `leftOp` `rightOp` positions to the right while inserting 0's to left; `leftOp &&& rightOp`, bitwise 'and', returns the result of taking the boolean 'and' operator position-wise; `leftOp ||| rightOp`, bitwise 'or', as 'and' but using the boolean 'or' operator; and `leftOp ^^^ leftOp`, bitwise xor, which is returns the result of the boolean 'xor' operator defined by the truth table in Table 5.5. · xor · exclusive or

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

Table E.1, E.2, and E.3 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,

Listing 5.27, floatDivisionRemainder.fsx:
Floating point division and remainder operators.

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

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

Listing 5.28, floatDivisionRemainderLossless.fsx:
Floating point division, truncation, and remainder is a lossless representation of a number.

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

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

Listing 5.29, floatDivisionByZero.fsx:
Floating point numbers include infinity and Not-a-Number.

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

However, the `float` type has limited precision, since there is only a finite number of numbers that can be stored in a float. E.g.,

Listing 5.30, floatImprecission.fsx:
Floating point arithmetic has finite precision.

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

That is, addition and subtraction associates to the left, hence the expression is interpreted as `(357.8 + 0.1) - 357.9`, and we see that we do not get the expected 0, 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, `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

Listing 5.31, upcaseChar.fsx:
Converting case by casting and integer arithmetic.

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

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

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

Listing 5.32, stringConcatenation.fsx:
Example of string concatenation.

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

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

Listing 5.33, stringIndexing.fsx:
String indexing using square brackets.

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

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

Listing 5.34, stringIndexingLength.fsx:
String length attribute and string indexing.

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

Since index counting starts at 0, and the string length is 7, then the index of the last character is 6. The is a long list of built-in functions in `System.String` for working with strings, some of which will be discussed in Chapter F.1.

The *dot notation* is an example of Structured programming, where technically speaking, the string `"abcdefg"` is an immutable *object* of *class* `string`, `[]` is an object *method*, and `Length` is a property. For more on object, classes, and methods see Chapter 20.

· dot notation
· object
· class
· method

Strings are compared letter by letter. For two strings to be equal, they must have the same length and all the letters must be identical. E.g., `"abs" = "absalon"` is false, while `"abs" = "abs"` is true. The `<>` operator is the boolean negation of the `=` operator, e.g., `"abs" <> "absalon"` is true, while `"abs" <> "abs"` is false. For the `<` , `<=`, `>`, and `>=` operators, the strings are ordered alphabetically, such that `"abs" < "absalon" && "absalon" < "milk"` is true, that is, the `<` operator on two strings is true, if the left operand should come before the right, when sorting alphabetically. The algorithm for deciding the boolean value of `leftOp < rightOp` is as follows: we start by examining the first character, and if `leftOp.[0]` and `rightOp.[0]` are different, then the `leftOp < rightOp` is equal to `leftOp.[0] < rightOp.[0]`. E.g., `"milk" < "abs"` is the same as `'m' < 'a'`, which is false, since the letter 'm' does not come before the letter 'a' in the alphabet, or more precisely, the codepoint of 'm' is not less than the codepoint of 'a'. If `leftOp.[0]` and `rightOp.[0]` are equal, then we move onto the next letter and repeat the investigation, e.g., `"abe" < "abs"` is true, since `"ab" = "ab"` is true and `'e' < 's'` is true. If we reach the end of either of the two strings, then the short is smaller than the larger, e.g., `"abs" < "absalon"` is true, while `"abs" < "abs"` is false. The `<=`, `>`, and `>=` operators are defined similarly.

5.7 Programmingintermezzo

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

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

we use the quadratic formula from elementary algebra,

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

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

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

```
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
```

Here 3 functions are defined as `determinant`, `postiveSolution`, 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# has adheres to two different syntax, regular and *ligthweight*. In the regular syntax, newlines and whitespaces are generally ignored, while in lightweight

· lightweight syntax

`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#.

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.

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, 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

An identifier is a name for a constant, an expression, or a type, and it is defined by the following EBNF:

`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#.

Listing 6.2: to do

```
ident = (letter | "_") {letter | dDigit | specialChar};
longIdent = ident | ident "." longIdent; (*no space around "."*)

longIdentOrOp = [longIdent "."] identOrOp; (*no space around "."*)
identOrOp =
  ident
  | "(" infixOp | prefixOp ")"
  | "(*)";

dDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
letter = Lu | Ll | Lt | Lm | Lo | Nl; (*e.g. "A", "B" ... and "a", "b", ...*)
specialChar = Pc | Mn | Mc | Cf; (*e.g., "_"*)

codePoint = ?Any unicode codepoint?;
Lu = ?Upper case letters?;
Ll = ?Lower case letters?;
Lt = ?Digraphic letters, with first part uppercase?;
Lm = ?Modifier letters?;
Lo = ?Gender ordinal indicators?;
Nl = ?Letterlike numeric characters?;
Pc = ?Low lines?;
Mn = ?Nonspacing combining marks?;
Mc = ?Spacing combining marks?;
Cf = ?Soft Hyphens?;
```

Thus, examples of identifiers are `a`, `theCharacter9`, `Next_Word`, `_tok`. Typically, only letters from the english alphabet are used as `letter`, and only _ is used for `specialChar`, but the full definition referes 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:

Listing 6.3: to do

```
expr = ...
  | expr ":" type (*type annotation*)
  | expr ";" expr (*sequence of expressions*)
  | "let" valueDefn "in" expr (*binding a value or variable*)
  | "let" ["rec"] functionDefn "in" expr (*binding a function or operator*)
  | "fun" argumentPats "->" expr (*anonymous function*)
  | expr "<-" expr (*assingment*)

type = ...
  | longIdent (*named such as "int"*)

valueDefn = ["mutable"] pat "=" expr;

pat = ...
  | "_" (*wildcard*)
  | ident (*named*)
  | pat ":" type (*type constraint*)
  | "(" pat ")" (*paranthesized*)

functionDefn = identOrOp argumentPats [":" type] "=" expr;
argumentPats = pat | pat argumentPats;
```

which will be discussed in the following. [1]

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,

Listing 6.4: to do

```
expr = ...
  | "let" valueDefn "in" expr (*binding a value or variable*)
```

The `let` bindings defines relations between patterns `pat` and expressions `expr` for many different purposes. Most often the pattern is an identifier`ident`, which *`let`* defines to be an alias of the expression `expr`. The pattern may also be defined to have specific type using the *:* lexeme and a named `type`. The `_` pattern is called the *wild card* pattern and, when it is in the value binding, then the expression is evaluated but the result is discarded. 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. For example, letting the identifier `p` be bound to the value `2.0` and using it in an expression is done as follows,

· `let`
· :
· wild card
· `mutable`
· lexically
· `in`

Listing 6.5, letValue.fsx:
The identifier p is used in the expression following the in keyword.

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

```
8.0
```

[1] Todo: **Mention special identifier '_', which means ignore.**

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,

Listing 6.6, letValueLF.fsx:
Newlines after in make the program easier to read.

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

```
9.0
```

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

Listing 6.7, letValueLightWeight.fsx:
Lightweight syntax does not require the in keyword, but expression must be aligned with the let keyword.

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

```
9.0
```

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

Listing 6.8, letValueLightWeightTypes.fsx:
Interactive mode also responds inferred types.

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

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

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

Listing 6.9, letValueTypeError.fsx:
Binding error due to type mismatch.

```
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
```

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.,

Listing 6.10, letValueSequence.fsx:
A value binding for a sequence of expressions.

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

```
2.0
9.0
```

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

Listing 6.11, letValueSequenceLightWeight.fsx:
A value binding for a sequence using lightweight syntax.

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

```
2.0
9.0
```

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., · scope

Listing 6.12, letValueScopeLower.fsx:
Redefining identifiers is allowed in lower scopes.

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

```
 4
```

Scopes are given levels, and scopes may be nested, where the nested scope has a level one lower than its parent.[2] F# distinguishes between the top and lower levels, and at the top level in the lightweight syntax, redefining values is not allowed, e.g.,

Listing 6.13, letValueScopeLowerError.fsx:
Redefining identifiers is not allowed in lightweight syntax at top level.

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

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

But using `begin` and `end` keywords, we create a *block* which acts as a *nested scope*, and then redefining · block · nested scope

[2] Todo: **Drawings would be good to describe scope**

is allowed, e.g.,

Listing 6.14, letValueScopeBlockAlternative2.fsx:
A block has lower scope level, and rebinding is allowed.

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

```
4
```

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

Listing 6.15, letValueScopeBlockAlternative3.fsx:
A block may be created using parentheses.

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

```
4
```

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 8.

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

Listing 6.16, letValueScopeBlockProblem.fsx:
Overshadowing hides the first binding.

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

```
4
4
```

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,[3]

Listing 6.17, letValueScopeBlock.fsx:
Blocks allow for the return to the previous scope.

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

```
 4
 3
```

[3]Todo: **spacing in lstinline 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.

4

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

Listing 6.18: to do

```
expr = ...
  | "let" functionDefn "in" expr (*binding a function or operator*)
```

Functions may also be recursive, which will be discussed in Chapter 8. An example in interactive mode is,

Listing 6.19, letFunction.fsx:
An example of a binding of an identifier and a function.

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

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

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

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

Listing 6.20: All types need most often not be specified.

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

or even just one of the arguments,

[4] Todo: **Remember to say something about interactive scripts and the ;; lexeme and scope**

Listing 6.21: Just one type is often enough for F# to infer the rest.

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

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

· operator
· operand

Listing 6.22, letFunctionLightWeight.fsx:
Lightweight syntax for function definitions.

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

```
1221.0
```

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

Listing 6.23, functionDeclarationGeneric.fsx:
Typesafety implies that a function will work for any type, and hence it is generic.

```
> 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 = ()
```

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 an 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,

Listing 6.24, identifiersExampleAdvance.fsx:
A function may contain sequences of expressions.

```
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
```

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,[5] · lexical scope

Listing 6.25, lexicalScopeNFunction.fsx:
Lexical scope means that $f(z) = 3x$ and not $4x$ at the time of calling.

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

```
6.0
```

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`. [6] Advice

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

[5]Todo: **Add a drawing or possibly a spell-out of lexical scope here.**
[6]Todo: **comment on dynamic scope and mutable variables.**

Listing 6.26, functionDeclarationAnonymous.fsx:
Anonymous functions are functions as values.

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

```
5
```

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

Listing 6.27, functionDeclarationAnonymousAdvanced.fsx:
Anonymous functions are often used as arguments for other functions.

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

```
18
```

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

Functions may be declared from other functions

Listing 6.28, functionDeclarationTupleCurrying.fsx:

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

```
15
6
```

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

Listing 6.29, functionDeclarationCurrying.fsx:

```
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
```

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

Listing 6.30, procedure.fsx:
A procedure is a function that has no return value, which in F# implies`()` as return value.

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

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

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

6.3 User-defined operators

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

Listing 6.31, addOperatorNFunction.fsx:

```
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
```

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

Listing 6.32: Grammar for infix and prefix lexemes

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

The precedence rules and associativity of user-defined operators follows the rules for which they share prefixes with built-in rules, see Table E.6. E.g., `.*`, `+++`, and `<+` are valid operator names for infix operators, they have precedence as ordered, and their associativity are all left. Using `~` as the first character in the definition of an operator makes the operator unary and will not be part of the name. Examples of definitions and use of operators are,

Listing 6.33, operatorDefinitions.fsx:

```
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
```

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.** In Chapter /refchap:oop we will discuss how to define type specific operators including prefix operators. Advice

Operators beginning with `*` must use a space in its definition, `( *` in order for it not to be confused with the beginning of a comment `(*`. [7]

[7] Todo: **this requires comments to be describe previously!**

6.4 The Printf function

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

Listing 6.34: to do

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

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

Listing 6.35: to do

```
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 placerholders 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.,

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

```
let pi = 3.1415192
let hello = "hello"
printf "An integer: %d\n" (int pi)
printf "A float %f on decimal form and on %e scientific form, 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\"\n\"%-8s\"\n" "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   "
```

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

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

Table 6.1: Printf placeholder string

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

Table 6.2: The family of printf functions.

examples of their use are,

Listing 6.37, printfExampleAdvance.fsx:

```
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"
```

The `%A` is special in that all built-in types including tuples, lists, and arrays to be discussed in Chapter 9 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 12. The function `failwithf` is used with exceptions, see Chapter 11 for more details. The function has a number of possible return value types, and for testing the *ignore* function ignores it all, e.g., `ignore (failwithf "\%d failed apples"3)` · ignore

6.5 Variables

The `mutable` in `let` bindings means that the identifier may be rebound to a new value using the lexeme, e.g.,[8] · <-

[8]Todo: **Discussion on heap and stack should be added here.**

Listing 6.38: to do

```
expr = ...
  | expr "<-" expr (*assingment*)
```

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,

· Mutable data
· variable

Listing 6.39, mutableAssignReassingShort.fsx:
A variable is defined and later reassigned a new value.

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

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,

· `<-`

Listing 6.40, mutableEqual.fsx:
Common error - mistaking = and <- lexemes for mutable variables. The former is the test operator, while the latter is the assignment expression.

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

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

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,

Listing 6.41, mutableAssignReassingTypeError.fsx:
Assignment type mismatching causes a compile time error.

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

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 that its previous value. For example,

Listing 6.42, mutableAssignIncrement.fsx:
Variable increment is a common use of variables.

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

```
5
6
```

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

Listing 6.43, mutableAssignIncrementEncapsulation.fsx:

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

```
1
2
3
```

[9] 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

· encapsulation

[9] Todo: **Explain why this works!**

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 6.25 defines a function using lexical scope and returns the number `6.0`, however, if `a` is made `mutable`, then the behaviour is different:

Listing 6.44, dynamicScopeNFunction.fsx:
Mutual variables implement dynamics scope rules. Compare with Listing 6.25.

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

```
8.0
```

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

Variables cannot be returned from functions, that is,

Listing 6.45, mutableAssignReturnValue.fsx:

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

```
0
```

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

Listing 6.46, mutableAssignReturnVariable.fsx:

```
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
```

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

Listing 6.47, mutableAssignReturnRefCell.fsx:

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

```
0
3
```

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, such as the following example demonstrates,[10] · side-effects

Listing 6.48, mutableAssignReturnSideEffect.fsx:

```
let updateFactor factor =
  factor := 2

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

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

```
6
```

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,

Listing 6.49, mutableAssignReturnWithoutSideEffect.fsx:

```
let updateFactor () =
  2

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

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

```
6
```

Here there can be no doubt in `multiplyWithFactor` that the value of '`a`' is changing. Side-effects do

[10] Todo: **Discuss side-effects!**

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.

[11]

[11] Todo: **Add something about mutable functions**

Chapter 7

In-code documentation

Documentation is a very important part of writing programs, since it is most unlikely, that you will be writing really obvious code. 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,

```
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.[1]

The F# compiler has an option for generating *Extensible Markup Language* (*XML*) files from scripts using the C# documentation comments tags[2]. 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 7.1. 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

[1] Todo: **lstlisting colors is bad.**

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

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

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

Listing 7.1, commentExample.fsx:
Code with XML comments.

```
/// 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>
///   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
```

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

Listing 7.2, Converting in-code comments to XML.

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

This results in an XML file with the following content,

Listing 7.3, An XML file generated by `fsharpc`.

```
<?xml version="1.0" encoding="utf-8"?>
<doc>
<assembly><name>commentExample</name></assembly>
<members>
<member name="M:CommentExample.solution(System.Double,System.Double,System
    .Double,System.Double)">
 <summary>Find x when 0 = ax^2+bx+c.</summary>
 <remarks>Negative 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,

· Hyper Text Markup Language
· HTML

Listing 7.4, Converting an XML file to HTML.

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

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 7.1. A full description of how to use `mdoc` is found here[3].

[3] `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 => x_+ = 0.7` printed to the console.

Requirements

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

Figure 7.1: Part of the HTML documentation as produce by `mdoc` and viewed in a browser.

Chapter 8

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 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,

Listing 8.1: to do

```
expr = ...
  | "if" expr "then" expr {"elif" expr "then" expr} ["else" expr] (*
     conditional*)
  | "while" expr "do" expr ["done"] (*while*)
  | "for" ident "=" expr "to" expr "do" expr ["done"] (* simple for expression
      *)
  | "let" ["rec"] functionDefn "in" expr (*binding a function or operator*)
```

8.1 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,[1] · for

Listing 8.2, count.fsx:
Counting from 1 to 10 using a for-loop.

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

val it : unit = ()
```

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

[1] Todo: **Is it clear enough that the body of the loop is repeated?**

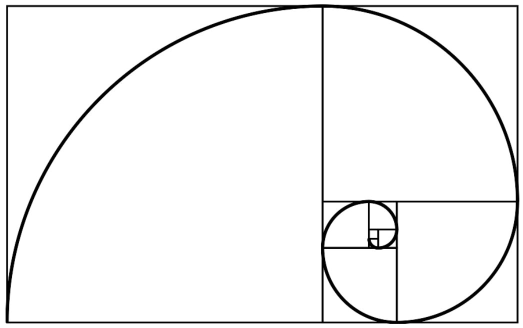

Figure 8.1: The Fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive Fibonacci numbers, and the curve in each square is the circular arc with radius of the square it is drawn in. Figure by Dicklyon `https://commons.wikimedia.org/w/index.php?curid=3730979`

`()` like the `printf` functions. The `for` and `while` loops follow the syntax,

Listing 8.3: U

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

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

Listing 8.4, countLightweight.fsx:
Counting from 1 to 10 using a `for`-loop.

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

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

A more complicated example is,

Problem 8.1:

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

The Fibonacci numbers is the series of numbers $1, 1, 2, 3, 5, 8, 13 \dots$, where the $\mathrm{fib}(n) = \mathrm{fib}(n-1) + \mathrm{fib}(n-2)$, and they are related to Golden spirals shown in Figure 8.1.[2] We could solve this problem with a `for`-loop as follows,

[2]Todo: **Should add to the figure, quadratic paper squares and area annotations to strenghten the relation to Fibonacci series.**

Listing 8.5, fibFor.fsx:
The n'th Fibonacci number as the sum of the previous 2 numbers, which are sequentially updated from 3 to n.

```
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
```

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

For the alternative `for`-loop, consider the problem,

Problem 8.2:

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

Prime numbers are integers divisible only be 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,

Listing 8.6, primeCheck.fsx:
Checking whether a given number has remainder zero after division by some low prime numbers.

```
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
```

In this example, the variable `i` runs through the elements of a list, which will be discussed in further detail in Chapter 9.

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 8.2 example, it would look somewhat like, · `while`

Listing 8.7, countWhile.fsx:
Count to 10 with a counter variable.

```
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
```

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,

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

```
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
```

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.,

Listing 8.9, forScopeError.fsx:
Lexical scope error. While rebinding is valid F# syntax, has little effect due to lexical scope.

```
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)
```

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.

8.2 Conditional expressions

Consider the task,

Problem 8.3:

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,

Listing 8.10: Using conditional expression to generate different strings.

```
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)
```

The grammar for conditional expressions is,

Listing 8.11: w

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

here 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

Listing 8.12: Lightweight syntax allows for making blocks of code by indentation in order to make code more for easy to read.

```
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"
```

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"`

8.3 Programming intermezzo

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

Problem 8.4:

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,

Listing 8.13, dec2bin.fsx:
Using integer division and remainder to convert any positive integer to binary form.

```
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
```

8.4 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, this 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

· recursive function

10 similarly to Listing 8.2 is,[3]

Listing 8.14, countRecursive.fsx:
Counting to 10 using recursion.

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

prt 1 10
```

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

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

Listing 8.15: U

```
expr = ...
  | "let" "rec" functionDefn "in" expr (*binding a function or operator*)
```

sing recursion to calculate the Fibonacci number as Listing 8.5.

[3]Todo: **A drawing showing the stack for the example would be good.**

Listing 8.16, fibRecursive.fsx:
The n'th Fibonacci number using recursive.

```
let rec fib n =
  if n < 1 then
    0
  elif n = 1 then
    1
  else
    fib (n - 1) + fib (n - 2)

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

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

Here we 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 15.1. Advice

[4]

[4]Todo: **Add short-cut if-then-else with and and or logical operators.**

Chapter 9

Ordered series of data

[1] F# is tuned to work with ordered series, and there are several built-in lists with various properties making them useful for different tasks. E.g.,

Listing 9.1, tuplesQuadraticEq.fsx:
Using tuples to gather values.

```
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
```

F# has 4 built-in list types: strings, tuples, lists, arrays, and sequences. Strings were discussed in Chapter 5, sequences will be discussed in Chapter 15. Here we will concentrate on tuples, lists, and arrays, and following this (simplified) syntax:

```
expr = ...
  | exprTuple (*tuple*)
  | "[" (exprSeq | rangeExpr) "]" (*list*)
  | "[|" (exprSeq | rangeExpr) "|]" (*array*)

exprTuple = expr | expr "," exprTuple;
exprSeq =  expr | expr ";" exprSeq;
rangeExpr = expr ".." expr [".." expr];
```

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

[1] Todo: **possibly add maps and sets as well.**

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.

9.1 Tuples

Tuples are unions of immutable types,

· tuple

```
expr = ...
  | exprTuple (*tuple*)

exprTuple = expr | expr "," exprTuple;
```

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

Listing 9.2, tuple.fsx:
Definition of a tuple.

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

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

The values `2`, `true`, and `"hello"` are *members*, and the number of elements of a tuple is its *length*. From the response of F# we see that the tuple is inferred to have the type `int * bool * string`, 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

Listing 9.3, tupleDeconstruction.fsx:
Definition of a tuple.

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

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

In this a function is defined that takes 1 argument, a 3-tuple, 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

Advice

definition in Listing 9.1. 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)`.

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')&& ('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.

Listing 9.4, tupleCompare.fsx:
Tuples are compared as strings are compared alphabetically.

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

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

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

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

The algorithm for deciding the boolean value of `(a1, a2)< (b1, b2)` is as follows: we start by examining the first elements, and if `la1` and `b1` are different, then the `(a1, a2)< (b1, b2)` is equal to `a1 < b1`. If `la1` and `b1` are equal, then we move onto the next letter and repeat the investigation. The `<=`, `>`, and `>=` operators are defined similarly.

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

Listing 9.5, tupleOfMutables.fsx:
A mutable change value, but the tuple defined by it does not refer to the new value.

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

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

However, tuples may be mutual 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.

Listing 9.6, fibTuple.fsx:
Calculating Fibonacci numbers using mutable tuple.

```
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
```

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 $\mathrm{fib}(n) = \mathrm{fib}(n-2) + \mathrm{fib}(n-1)$ and the rearrangement of memory to hold the new values $\mathrm{fib}(n)$ and $\mathrm{fib}(n-1)$ based on the old values $\mathrm{fib}(n-2)+\mathrm{fib}(n-1)$. 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

9.2 Lists

Lists are unions of immutable values of the same type and have a more flexible structure than tuples. Its grammar follows *computation expressions*, which is very rich and shared with arrays and sequences, and we will delay a discussion on most computation expressions to Section 15.1, and here just consider a subset of the grammar:

· list

· computation expressions

```
expr = ...
  | "[" (exprSeq | rangeExpr) "]" (*list*)

exprSeq =  expr | expr ";" exprSeq;
rangeExpr = expr ".." expr [".." expr];
```

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]`.

· range expression

Lists may be indexed and concatenated much like strings, e.g.,

Listing 9.7, listIndexing.fsx:
Examples of list concatenation, indexing.

```
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
```

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 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

Listing 9.8, listSlicing.fsx:
Examples of list slicing. Compare with Listing 5.33.

```
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']
```

The basic properties and members of lists are summarized in Table 9.1. In addition, lists have many other built-in functions, such as functions for converting lists to arrays and sequences,

Listing 9.17, listConversion.fsx:
The `List` module contains functions for conversion to arrays and sequences.

```
let lst = ['a' .. 'c']
let arr = List.toArray lst
let sq = List.toSeq lst
printfn "%A, %A, %A" lst arr sq
```

```
['a'; 'b'; 'c'], [|'a'; 'b'; 'c'|], ['a'; 'b'; 'c']
```

These and more will be discussed in Chapter F and Part III.[2]

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

Listing 9.18, listMultidimensional.fsx:
A ragged multidimensional list, built as lists of lists, and its indexing.

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

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

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

9.3 Arrays

One dimensional arrays or just arrays for short are mutable lists of the same type and follow a similar syntax as lists. Its grammar follows *computation expressions*, which will be discussed in Section 15.1. Here we consider a subset of the grammar:

· computation expressions

```
expr = ...
  | "[|" (exprSeq | rangeExpr) "|]" (*array*)

exprSeq =  expr | expr ";" exprSeq;
rangeExpr = expr ".." expr [".." expr];
```

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

[2]Todo: **Add description of prepend and concatenation operator for lists.**

<table>
<tr><th>Function name</th><th>Example</th><th>Description</th></tr>
<tr><td><code>Length</code></td><td>Listing 9.9:
<code>> [1; 2; 3].Length;;</code>
<code>val it : int = 3</code>
<code>> let a = [1; 2; 3]</code>
<code>   in a.Length;;</code>
<code>val it : int = 3</code></td><td>The number of elements in a list</td></tr>
<tr><td><code>List.Empty</code></td><td>Listing 9.10:
<code>> let a : int list =</code>
<code>   List.Empty;;</code>

<code>val a : int list =</code>
<code>   []</code>

<code>> let b = List<int>.</code>
<code>   Empty;;</code>

<code>val b : int list =</code>
<code>   []</code></td><td>An empty list of specified type</td></tr>
<tr><td><code>IsEmpty</code></td><td>Listing 9.11:
<code>> [1; 2; 3].IsEmpty</code>
<code>   ;;</code>
<code>val it : bool =</code>
<code>   false</code>
<code>> let a = [1; 2; 3]</code>
<code>   in a.IsEmpty;;</code>
<code>val it : bool =</code>
<code>   false</code></td><td>Compare with the empty list</td></tr>
<tr><td><code>Item</code></td><td>Listing 9.12:
<code>> [1; 2; 3].Item 1;;</code>
<code>val it : int = 2</code>
<code>> let a = [1; 2; 3]</code>
<code>   in a.Item 1;;</code>
<code>val it : int = 2</code></td><td>Indexing</td></tr>
<tr><td><code>Head</code></td><td>Listing 9.13:
<code>> [1; 2; 3].Head;;</code>
<code>val it : int = 1</code>
<code>> let a = [1; 2; 3]</code>
<code>   in a.Head;;</code>
<code>val it : int = 1</code></td><td>The first element in the list. Exception if empty.</td></tr>
<tr><td><code>Tail</code></td><td>Listing 9.14:
<code>> [1; 2; 3].Tail;;</code>
<code>val it : int list =</code>
<code>   [2; 3]</code>
<code>> let a = [1; 2; 3]</code>
<code>   in a.Tail;;</code>
<code>val it : int list =</code>
<code>   [2; 3]</code></td><td>The list except its first element</td></tr>
</table>

Listing 9.19, arrayCreation.fsx:
Creating arrays with a syntax similarly to lists.

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

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

Listing 9.20, arrayReassign.fsx:
Arrays are mutable in spite the missing `mutable` keyword.

```
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
```

Notice that in spite the missing `mutable` keyword, the function `square` still had the *side-effect* of squaring alle entries in `A`. · side-effect

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., · slicing

Listing 9.21, arraySlicing.fsx:
Examples of array slicing. Compare with Listing 9.8 and Listing 5.33.

```
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'|]
```

As illustrated, the missing start or end index implies from the first or to the last element.

Arrays can be converted to lists and sequences by,

Listing 9.22, arrayConversion.fsx:
The `Array` module contains functions for conversion to lists and sequences.

```
let arr = [|'a' .. 'c'|]
let lst = Array.toList arr
let sq = Array.toSeq arr
printfn "%A, %A, %A" arr lst sq
```

```
[|'a'; 'b'; 'c'|], ['a'; 'b'; 'c'], seq ['a'; 'b'; 'c']
```

There are quite a number of built-in procedures for all arrays many which will be discussed in Chapter F.

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

Listing 9.23, arrayJagged.fsx:
An array of arrays. When row lengths are of non-equal elements, then it is a Jagged array.

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

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

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

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

Listing 9.24, arrayJaggedSquare.fsx:
A rectangular array.

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

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

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

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

Notice, the `for-in` cannot be used in `pownArray`, e.g., `for row in arr do for elm in row do elm <- pown elm p done done} since the iterator value \lstinlineelm!` 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,

Listing 9.25, array2D.fsx:
Creating a 3 by 4 rectangular arrays of intigers.

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

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

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

Listing 9.26, array2DSlicing.fsx:
Examples of Array2D slicing. Compare with Listing 9.25.

```
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]]
```

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. [3]

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

[3]Todo: **Array2D.ToString produces `[[ ... ]]` and not `[|[| ... |]|]`, which can cause confusion.**

Chapter 10

Testing programs

A software bug is an error in a computer program that causes it to produce an incorrect result or behave in an unintended manner. The term bug was used by Thomas Edison in 1878[1], but made popular in computer science by Grace Hopper, who found a moth interferring with the electronic circuits of the Harward Mark II electromechanical computer and coined the term *bug* for errors in computer programs. The original bug is shown in Figure 10.1. Software is everywhere, and errors therein have huge economic impact on our society and can threaten lives[2].

· bug

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

· functionality

Functionality: Does the software compile and run without internal errors. Does it solve the problem, it was intended to solve? E.g., does the route planning software finde a suitable route from point a to b?

· reliability

Reliability: Does the software work reliably over time? E.g., does the route planning software work

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

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

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

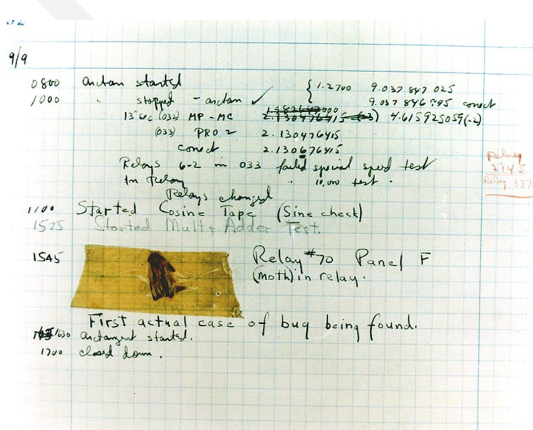

Figure 10.1: The first computer bug caught by Grace Hopper, U.S. Naval Historical Center Online Library Photograph NH 96566-KN.

in case of internet dropouts?

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

Efficiency: How many computer and human resources does the software require? E.g., does it take milliseconds or hours to find a requested route? Can the software run on a mobile platform with limited computer speed and memory? · efficiency

Maintainability: In case of the discovery of new bugs, is it easy to test and correct the software? Is it easy to extend the software with new functionality? E.g., is it easy to update the map with updated roadmaps and new information? Can the system be improved to work both for car drivers and bicyclists? · maintainability

Portability: Is it easy to port the software to new systems such as new server architecture and screen sizes? E.g., if the routing software originally was written for IOS devices, will it be easy to port to Android systems? · portability

The above mentioned concepts are ordered based on the requirements of the system. Functionality and reliability ares perhaps the most important concepts, since if the software does not solve the specified problem, then the software designing process has failed. However, many times the problem definition will evolve along with the software development process. But as a bare minimum, the software should run without internal errors and not crash under well defined set of circumstances. Further, it is often the case, that software designed for the general public requires a lot of attention to the usability of the software, since in many cases non-experts are expected to be able to use the software little or no prior training. On the other hand, software used internally in companies will be used by a small number of people, who become experts in using the software, and it is often less important that the software is easy to understand by non-experts. An example is text processing software Microsoft Word versus Gnu Emacs and LaTeX. Word is designed to be used by non-experts for small documents such as letters and notes, and relies heavily on interfacing with the system using click-interaction. On the other hand, Emacs and LaTeX are for experts for longer and professionally typeset documents, and relies heavily on keyboard shortcuts and text-codes for typesetting document entities.

The purpose of *software testing* is to find bugs. For errors found we engage in *debugging*, which is the process of diagnosing and correcting bugs. Once we have a failed software test, i.e., one that does not find any bugs, then we have strengthened our belief in the software, but it is important to note, that software testing and debugging rarely removes all bugs, and with each correction or change of software, there is a fair chance of introducing new bugs. It is not exceptional, that the software testing the software is as large as the original.

· software testing

· debugging

In this chapter, we will focus on two approaches to software testing, which emphasizes functionality: *white-box* and *black-box testing*. An important concept in this context is *unit testing*, where the program is considered in smaller pieces, called units, and for which accompanying programs for testing can be made, which tests these units automatically. Black-box testing considers the problem formulation and the program interface, and can typically be written early in the software design phase. In contrast, white-box testing considers the program text, and thus requires the program to be available. Thus there is a tendency for black-box test programs to be more stable, while white-box testing typically is developed incrementally along side the software development.

· white-box testing

· black-box testing

· unit testing

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

Problem 10.1:

Given any date in the Gregorian calendar, calculate the day of week.

Facts about dates in the Gregorian calendar are:

- combinations of dates and weekdays repeat themselves every 400 years;
- the typical length of the months Januar, February, ... follow the knucle rule, i.e., January belongs to the index knuckle, February to the space between the index and the middle finger, and August restarts or starts on the other hand. All knuckle months have 31 days, all spacing months have 30 days except February, which has 29 days on leap years and 28 days all other years.
- A leap year is a multiplum of 4, except if it is also a multiplum of 100 but not of 400.

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

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

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

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

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

10.1 White-box testing

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

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

1. Decide which are the units to test: The program shown in Listing 10.1 has 3 functions, and we will consider these each as a unit, but we might as well just have chosen `date2Day` as a single unit. The important part is that the union of units must cover the whole program text, and since `date2Day` calls both `januaryFirstDay` and `sum`, designing test cases for the two later is

superfluous. However, we may have to do this anyway, when debugging, and we may choose at a later point to use these functions separately, and in both cases we will be able to reuse the testing of the smaller units.

2. Identify branching points: The function `januaryFirstDay` has no branching function, `sum` has one, and depending on the input values two paths through the code may be used, and `date2Day` has one, where the number of days in February is decided. Note that in order to test this, our test-date must be March 1 or later. In this example, there are only examples of `if`-branch points, but they may as well be loops and pattern matching expressions. In the following code, the branch points have been given a comment and a number,

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

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

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

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

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

Unit	Branch	Condition	Input	Expected output
`januaryFirstDay`	0	-	2016	5
`sum`	1	`0 <= j && j < lst.Length`		
	1a	`true && true`	`[1; 2; 3] 1`	3
	1b	`false && true`	`[1; 2; 3] -1`	0
	1c	`true && false`	`[1; 2; 3] 10`	0
	1d	`false && false`	-	-

`date2Day`	1	`(y % 4 = 0)`
		`&& ((y % 100 <> 0)`
		`|| (y % 400 = 0))`		

	-	`true && (true || true)`	-	-
	1a	`true && (true || false)`	`8 9 2016`	Thursday
	1b	`true && (false || true)`	`8 9 2000`	Friday
	1c	`true && (false || false)`	`8 9 2100`	Wednesday
	-	`false && (true || true)`	-	-
	1d	`false && (true || false)`	`8 9 2015`	Tuesday
	-	`false && (false || true)`	-	-
	-	`false && (false || false)`	-	-

The impossible cases have been intentionally blank, e.g., it is not possible for $j < 0$ and $j > n$ for some positive value n.

4. Write a program, that test all these cases and checks the output, e.g.,

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

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

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

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

```
White-box testing of date2Day.fsx
  Unit: januaryFirstDay
    Branch: 0 - true
  Unit: sum
    Branch: 1a - true
    Branch: 1b - true
    Branch: 1c - true
  Unit: date2Day
    Branch: 1a - true
    Branch: 1b - true
    Branch: 1c - true
    Branch: 1d - true
```

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

After the white-box testing has failed to find errors in the program, we have some confidence in the program, since we have run every line at least once. It is, however, in no way a guarantee, that the program is error free, which is why white-box testing is often accompanied with black-box testing to be described next.

10.2 Back-box testing

In black-box testing the program is considered a black box, and no knowledge is required about how a particular problem is solved, in fact, it is often useful not to have that knowledge at all. It is rarely possible to test all input to a program, so in black-box testing, the solution is tested for typical and extreme cases based on knowledge of the problem. The procedure is as follows:

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

Make an overall description of the tests to be performed and their purpose:

1 a consecutive week, to ensure that all weekdays are properly returned

2 two set of consecutive days across boundaries that may cause problems: across a new year, across a regular month boundary.

3 a set of consecutive days across February-March boundaries for a leap and non-leap year

4 four dates after february in a non-multiplum-of-100 leap year and in a non-leap year, a multiplum-of-100-but-not-of-400 leap year, and a multiplum-of-100-but-and-of-400 leap year.

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

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

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

Write a program executing the tests:

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

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

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

```
Black-box testing of date2Day.fsx
  1. A complete week
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
    test 5 - true
    test 6 - true
    test 7 - true
  2. Across boundaries
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
  3. Across feburary bondary
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
    test 5 - true
  4. Leap years
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
```

Notice how the program has been made such that it is almost a direct copy of the table, produced in the previous step.

Advice

A black-box test is a statement of what a solution should fulfill for a given problem. Hence, **it is a good idea to make a black-box test early in the software design phase, in order to clarify the requirements for the code to be developed, and take an outside view of the code prior to developing it.**

After the black-box testing has failed to find errors in the program, we have some confidence in the program, since from a user's perspective, the program produces sensible output in many casses. It is, however, in no way a guarantee, that the program is error free.

10.3 Debugging by tracing

· debugging

· mockup code

· tracing

· hand tracing

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

Consider the program,

Listing 10.1: The greatest common divisor of 2 integers.

```
1  let rec gcd a b =
2    if a < b then
3      gcd b a
4    elif b > 0 then
5      gcd b (a % b)
6    else
7      a
8
9  let a = 10
10 let b = 15
11 printfn "gcd %d %d = %d" a b (gcd a b)
```

· environment

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

$$E_0:$$
$$\text{gcd} \rightarrow \big((a,b), \text{gcd-body}, \varnothing\big)$$

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

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

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

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

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

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

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

Hence, we update our trace as,

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

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

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

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

E_0 :

gcd $\to \big((a,b), \text{gcd-body}, \varnothing\big)$

$a \to 10$

$b \to 15$

line 11: gcd a b $\to$?

E_1 :$\big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big)$

line 3: gcd b a $\to$?

E_2 :$\big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big)$

line 5: a % b $\to$ 5

line 5: gcd b (a % b) $\to$?

E_3 :$\big((a \to 10, b \to 5), \text{gcd-body}, \varnothing\big)$

line 5: a % b $\to$ 0

line 5: gcd b (a % b) $\to \cancel{?}\,5$

$\cancel{E_4}$:$\big((a \to 5, b \to 0), \text{gcd-body}, \varnothing\big)$

Now line 5 in E_3 is also a return point of `gcd`, hence we scratch E_3 and change the question mark in E_2 to 5,

E_0 :

gcd $\to \big((a,b), \text{gcd-body}, \varnothing\big)$

$a \to 10$

$b \to 15$

line 11: gcd a b $\to$?

E_1 :$\big((a \to 10, b \to 15), \text{gcd-body}, \varnothing\big)$

line 3: gcd b a $\to$?

E_2 :$\big((a \to 15, b \to 10), \text{gcd-body}, \varnothing\big)$

line 5: a % b $\to$ 5

line 5: gcd b (a % b) $\to \cancel{?}\,5$

$\cancel{E_3}$:$\big((a \to 10, b \to 5), \text{gcd-body}, \varnothing\big)$

line 5: a % b $\to$ 0

line 5: gcd b (a % b) $\to \cancel{?}\,5$

$\cancel{E_4}$:$\big((a \to 5, b \to 0), \text{gcd-body}, \varnothing\big)$

and likewise for E_2 and E_1:

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

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

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

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

F# uses lexical scope, which implies that besides function arguments, we also at times need to consider the environment at place of writing. E.g., for the program

Listing 10.2: Example of lexical scope and closure environment.

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

```
4     let a = 4.0
5     f x
6   printfn "%A" (testScope 2.0)
```

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

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

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

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

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

$$
\begin{aligned}
&E_0:\\
&\quad \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big)\\
&\quad \text{line 6: testScope 2.0} \rightarrow ?\\
&E_1: \big(x \rightarrow 2.0, \text{testScope-body}, \varnothing\big)\\
&\quad a \rightarrow 3.0\\
&\quad \text{f} \rightarrow \big(z, \text{a * x}, (a \rightarrow 3.0, x \rightarrow 2.0)\big)\\
&\quad a \rightarrow 4.0\\
&\quad \text{line 5: f x} \rightarrow ?\\
&E_2: \big(z \rightarrow 10.0, \text{a * x}, (a \rightarrow 3.0, x \rightarrow 2.0\big)
\end{aligned}
$$

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

$$
\begin{aligned}
&E_0:\\
&\quad \text{testScope} \rightarrow \big(x, \text{testScope-body}, \varnothing\big)\\
&\quad \text{line 6: testScope 2.0} \rightarrow \cancel{?}\ 6.0\\
&\quad \text{line 6: stdout} \rightarrow \text{"6.0"}\\
&\cancel{E_1}: \big(x \rightarrow 2.0, \text{testScope-body}, \varnothing\big)\\
&\quad a \rightarrow 3.0\\
&\quad \text{f} \rightarrow \big(z, \text{a * x}, (a \rightarrow 3.0, x \rightarrow 2.0)\big)\\
&\quad a \rightarrow 4.0\\
&\quad \text{line 5: f x} \rightarrow \cancel{?}\ 6.0\\
&\cancel{E_2}: \big(z \rightarrow 10.0, \text{a * x}, (a \rightarrow 3.0, x \rightarrow 2.0\big)
\end{aligned}
$$

For mutable bindings, i.e., variables, the scope is dynamic. For this we need the concept of storage, i.e., for the the program

Listing 10.3: Example of dynamic scope and closure environment.

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

We add a storage area to our hand tracing, e.g.,line 6,

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

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

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

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

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

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

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

Hence,

$$
\begin{array}{ll}
\text{Store}: & \\
& \alpha_1 \to \cancel{3.0}\ 4.0 \\
E_0: & \\
& \text{testScope} \to \big(x, \text{testScope-body}, \varnothing\big) \\
& \text{line 6: testScope 2.0} \to ? \\
E_1: & \big(x \to 2.0, \text{testScope-body}, \varnothing\big) \\
& a \to \alpha_1 \\
& \text{f} \to \big(z, \text{a * x}, (a \to \alpha_1, x \to 2.0)\big) \\
& \text{line 5: f x} \to ? \\
E_2: & \big(z \to 10.0, \text{a * x}, (a \to \alpha_1, x \to 2.0\big)
\end{array}
$$

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

$$
\begin{array}{ll}
\text{Store}: & \\
& \alpha_1 \to \cancel{3.0}\ 4.0 \\
E_0: & \\
& \text{testScope} \to \big(x, \text{testScope-body}, \varnothing\big) \\
& \text{line 6: testScope 2.0} \to \cancel{?}\ 8.0 \\
& \text{line 6: stdout} \to \text{"8.0"} \\
\cancel{E_1}: & \big(x \to 2.0, \text{testScope-body}, \varnothing\big) \\
& a \to \alpha_1 \\
& \text{f} \to \big(z, \text{a * x}, (a \to \alpha_1, x \to 2.0)\big) \\
& \text{line 5: f x} \to \cancel{?}\ 8.0 \\
\cancel{E_2}: & \big(z \to 10.0, \text{a * x}, (a \to \alpha_1, x \to 2.0\big)
\end{array}
$$

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

4

---

4 Todo: **Should add something about hypotheses about sources of bugs possibly tied together with the use of `printfln`.**

Chapter 11

Exceptions

Exceptions are runtime errors, which may be handled gracefully by F#. Exceptions are handled by the !try! keyword both in expressions. E.g., Integer division by zero raises and exception, but it may be handled in a script as follows,

Listing 11.1, exceptionDivByZero.fsx:
A division by zero is caught and a default value is returned.

```
let div enum denom =
   try
      enum / denom
   with
      | :? System.DivideByZeroException -> System.Int32.MaxValue

printfn "3 / 1 = %d" (div 3 1)
printfn "3 / 0 = %d" (div 3 0)
```

```
3 / 1 = 3
3 / 0 = 2147483647
```

The `try` expressions have the following syntax,

```
expr = ...
  | "try" expr "with" ["|"] rules (*exception*)
  | "try" expr "finally" expr; (*exception with cleanup*)

rules = rule | rule "|" rules;
rule = pat ["when" expr] "->" expr;
```

Exceptions are a basic-type called `exn`, and F# has a number of built-in, see Table 11.1. The programs may define new exceptions using the syntax,

```
"exception" ident of typeTuple (*exception definition*)
typeTuple = type | type "*" typeTuple;
```

and any exceptions may be *raised* using the functions `failwith`, `invalidArg`, `raise`, and `reraise`. An example of raising an exception with the `raise` function is,

· raise an exception

Attribute	Description
`System.ArithmeticException`	Failed arithmetic operation.
`System.ArrayTypeMismatchException`	Failed attempt to store an element in an array failed because of type mismatch.
`System.DivideByZeroException`	Failed due to division by zero.
`System.IndexOutOfRangeException`	Failed to access an element in an array because the index is less than zero or equal or greater than the length of the array.
`System.InvalidCastException`	Failed to explicitly convert a base type or interface to a derived type at run time.
`System.NullReferenceException`	Failed use of a `null` reference was used, since it required the referenced object.
`System.OutOfMemoryException`	Failed to use `new` to allocate memory.
`System.OverflowException`	Failed arithmetic operation in a checked context which caused an overflow.
`System.StackOverflowException`	Failed use of the internal stack caused by too many pending method calls, e.g., from deep or unbounded recursion.
`System.TypeInitializationException`	Failed initialization of code for a type, which was not caught.

Table 11.1: Built-in exceptions.

Listing 11.2, exceptionDefinition.fsx:
A user-defined exception is raised but not caught by outer construct.

```
exception DontLikeFive of string

let picky a =
  if a = 5 then
    raise (DontLikeFive "5 sucks")
  else
    a

printfn "picky %A = %A" 3 (picky 3)
printfn "picky %A = %A" 5 (picky 5)
```

```
picky 3 = 3
FSI_0001+DontLikeFive: Exception of type 'FSI_0001+DontLikeFive' was
   thrown.
  at FSI_0001.picky (Int32 a) <0x66f3f58 + 0x00057> in <filename unknown
     >:0
  at <StartupCode$FSI_0001>.$FSI_0001.main@ () <0x66f31a0 + 0x0017f> 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 + 0
     x000a1> in <filename unknown>:0
Stopped due to error
```

Here an exception called `DontLikeFive` is defined, and it is raised in the function `picky`. When run, F# stops at run-time after the program has raised the exception with a long description of the reason including the name of the exception. Exceptions include messages, and the message for `DontLikeFive` is of type `string`. This message is passed to the `try` expression and may be processed as e.g.,

Listing 11.3, exceptionDefinitionNCatch.fsx:
Catching a user-defined exception.

```
exception DontLikeFive of string

let picky a =
  if a = 5 then
    raise (DontLikeFive "5 sucks")
  else
    a

try
  printfn "picky %A = %A" 3 (picky 3)
  printfn "picky %A = %A" 5 (picky 5)
with
  | DontLikeFive msg -> printfn "Exception caught with message: %s" msg
```

```
picky 3 = 3
Exception caught with message: 5 sucks
```

Note that the type of `picky` is `a:int -> int` because its argument is compared with an integer in the conditional statement. This contradicts the typical requirements for `if` statements, where every branch has to return the same type. However, any code that explicitly raises exceptions are ignored, and the type is inferred by the remaining branches.

The `failwith : string -> exn` function takes a string and raises the built-in `System.Exception` exception,

Listing 11.4, exceptionFailwith.fsx:
An exception raised by `failwith`.

```
if true then failwith "hej"
```

```
System.Exception: hej
  at <StartupCode$FSI_0001>.$FSI_0001.main@ () <0x676f158 + 0x00037> 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 + 0
     x000a1> in <filename unknown>:0
Stopped due to error
```

To catch the `failwith` exception, there are two choices, either use the `:?` or the `Failure` pattern. the `:?` pattern matches types, and we can match with the type of `System.Exception` as,

Listing 11.5, exceptionSystemException.fsx:
Catching a `failwith` exception using type matching pattern.

```
let _ =
  try
    failwith "Arrrrg"
  with
    :? System.Exception -> printfn "So failed"
```

```
/Users/sporring/repositories/fsharpNotes/src/exceptionSystemException.fsx
    (5,5): warning FS0067: This type test or downcast will always hold


/Users/sporring/repositories/fsharpNotes/src/exceptionSystemException.fsx
    (5,5): warning FS0067: This type test or downcast will always hold
So failed
```

However, this gives annoying warnings, since F# internally is built such that all exception matches the type of `System.Exception`. Instead it is better to either match anything,

Listing 11.6, exceptionMatchWildcard.fsx:
Catching a `failwith` exception using the wildcard pattern.

```
let _ =
  try
    failwith "Arrrrg"
  with
    _ -> printfn "So failed"
```

```
So failed
```

or use the built in `Failure` pattern,

Listing 11.7, exceptionFailure.fsx:
Catching a `failwith` exception using the `Failure` pattern.

```
let _ =
  try
    failwith "Arrrrg"
  with
    Failure msg ->
      printfn "The castle of %A" msg
```

```
The castle of "Arrrrg"
```

Notice how only the `Failure` pattern allows for the parsing of the message given to `failwith` as argument.

The `invalidArg` takes 2 strings and raises the built-in `ArgumentException`

Listing 11.8, exceptionInvalidArg.fsx:
An exception raised by `invalidArg`.

```
if true then invalidArg "a" "is too much 'a'"
```

```
System.ArgumentException: is too much 'a'
Parameter name: a
  at <StartupCode$FSI_0001>.$FSI_0001.main@ () <0x666f1f0 + 0x0005b> 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 + 0
     x000a1> in <filename unknown>:0
Stopped due to error
```

This would be caught by type matching as,

Listing 11.9, exceptionInvalidArgNCatch.fsx:
Catching the exception raised by `invalidArg`.

```
let _ =
  try
    invalidArg "a" "is too much 'a'"
  with
    :? System.ArgumentException -> printfn "Argument is no good!"
```

```
Argument is no good!
```

The `try` construction is typically used to gracefully handle exceptions, but there are times, where you may want to pass on the bucket, so to speak, and reraise the exception. This can be done with the `reraise`.

Listing 11.10, exceptionReraise.fsx:
Reraising an exception.

```
let _ =
  try
    failwith "Arrrrg"
  with
    Failure msg ->
      printfn "The castle of %A" msg
      reraise()
```

```
The castle of "Arrrrg"
System.Exception: Arrrrg
  at <StartupCode$FSI_0001>.$FSI_0001.main@ () <0x6745e88 + 0x00053> in <
     filename unknown>:0
Stopped due to error
```

The `reraise` function is only allowed to be the final call in the expression of a `!`with! rule.

At exceptions, it is not always obvious what should be returned. E.g., in the Listing 11.1, the exception is handled gracefully, but the return value is somewhat arbitrarily chosen to be the largest possible integer, which is still far from infinity, which is the correct result. Instead we could use the ***option*** type. The `option` type is a wrapper, that can be put around any type, and which extends the type with the special value `None`. All other values are preceded by the `Some` identifier. E.g., to rewrite Listing 11.1 to correctly represent the non-computable value, we could write · `option`

Listing 11.11, exceptionDivByZeroOptionType.fsx:
Option types can be used, when the value in case of exceptions is unclear.

```
> let div enum denom =
-   try
-     Some (enum / denom)
-   with
-     | :? System.DivideByZeroException -> None;;

val div : enum:int -> denom:int -> int option

>
- let a = div 3 1;;

val a : int option = Some 3

> let b = div 3 0;;

val b : int option = None
```

The value of an option type can be extracted by and tested for by its member function, `IsNone`, `IsSome`, and `Value`, e.g.,

Listing 11.12, option.fsx:
Simple operations on option types.

```
Some 3 <null>
3 false true
```

In the `try`-`finally`, the `finally` expression is always executed, e.g.,

Listing 11.13, exceptionFinally.fsx:
The `finally` expression in `try`-`finally` will always be executed.

```
Finally expression will always be executed.
System.Exception: True
  at <StartupCode$FSI_0001>.$FSI_0001.main@ () <0x6745328 + 0x0003f> 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 + 0
      x000a1> in <filename unknown>:0
Stopped due to error
```

This is useful for cleaning up, e.g., closing files etc. which we will discuss in Chapter 12. The only way to combine `try`-`finally` with `try`-`with` is to nest the expression inside each other.

Chapter 12

Input and Output

[1] An important part of programming is handling data. A typical source of data are hard-coded bindings and expressions from libraries or the program itself, and the result is often shown on a screen either as text output on the console. This is a good starting point, when learning to program, and one which we have relied heavily upon in this book until now. However, many programs require more: We often need to ask a user to input data via, e.g., typing text on a keyboard, clicking with a mouse, striking a pose in front of a camera. We also often need to load and save data to files, retrieve and deposit information from the internet, and visualize data as graphically, as sounds, or by controlling electrical appliances. Graphical user interfaces will be discussed in Chapter 13, and here we will concentrate on working with the console, with files, and with the general concept of streams.

File and stream input and output are supported via libraries built-in classes. The `printf` family of functions is defined in the `.Printf` module of the `Fsharp.Core'` namespace, and it was discussed in Chapter 6.4, and will not be discussed here. What we will concentrate on is interaction with the console through the `System.Console` class and the `System.IO` namespace.

A *file* on a computer is a resource used to store data in and retrieve data from. Files are often associated with a physical device, such as a harddisk, but can also be a virtual representation in memory. Files are durable, such that other programs can access them independently, given certain rules for access. A file has a name, a size, and a type, where the type is related to the basic unit of storage such as characters, bytes, and words, (`char`, `byte`, and `int32`). Often data requires a conversion from the internal format to and from the format stored in the file. E.g., floating point numbers are sometimes converted to ASCII using `fprintf` in order to store them to file in a human readable form, and interpreted from ASCII when retrieving them at a later point from file. Files have a low-level structure and representation, which varies from device to device, and the low-level details are less relevant for the use of the file, and most often hidden for the user. Basic operations on files are creation, opening, reading from, writing to, closing, and deleting files. · file

A *stream* is similar to files in that they are used to store data in and retrieve data from, but streams only allow for handling of data one element at a time like the readout of a thermometer: we can make temperature readings as often as we like, producing a history of temperatures, but we cannot access the future. Hence, streams are in principle without an end, and thus have infinite size, and data from streams are programmed locally by considering the present and previous elements, while data from files may be considered a stream but also allow for global operations on all the file's data. · stream

[1] Todo: **Work in progress!**

12.1 Interacting with the console

[2] From a programming perspective, then the console is a stream: The program may send new data to the console, but cannot return to previously sent data and make changes. Likewise, the program may retrieve input from the user, but cannot go back and ask the user to have inputted something else. The console uses 3 built-in streams in `System.Console`,,,

Stream	Description
`stdout`	Standard output stream used by `printf` and `printfn`.
`stderr`	Standard error stream used to display warnings and errors by Mono.
`stdin`	Standard input stream used to read keyboard input.

On the console, the standard output and error streams are displayed as text, and it is typically not possible to see a distinction between them. However, command-line interpreters such as Bash can, and it is possible from the command-line to filter output from programs according to these streams. However, a further discussion on this is outside the scope of this text. In `System.Console` there are many functions supporting interaction with the console, and the most important ones are,

Function	Description
`Write string`	Write to the console. E.g., `System.Console.Write "Hello world."`.
`WriteLine string`	As `Write` but followed by newline, e.g., `System.Console.WriteLine "Hello world."`.
`Read ()`	Read the next key from the keyboard blocking execution as long, e.g., `System.Console.Read ()`.
`ReadKey ()`	As `Read` but writing the key to the console as well, e.g. , `System.Console.ReadKey ()`.
`ReadLine ()`	Read the next sequence of characters until newline from the keyboard, e.g. , `System.Console.ReadLine ()`.

Notice that you must supply the empty argument `()`, in order to run most of the functions instead of referring to them as values. Note also, that

Listing 12.1: Interacting with a user with `ReadLine` and `WriteLine`.

```
System.Console.WriteLine "To perform the multiplication of a and b"
System.Console.Write "Enter a: "
let a = float (System.Console.ReadLine ())
System.Console.Write "Enter b: "
let b = float (System.Console.ReadLine ())
System.Console.WriteLine ("a * b = " + string (a * b))
```

An example dialogue is,

```
To perform the multiplication of a and b
Enter a: 2.3
Enter b: 4.5
a * b = 10.35
```

The `Write` functions has less functionality than the `printf` family, and **for writing to the console, `printf` is to be preferred.** Advice

[2]Todo: **Spec-4.0 Section 18.2.9**

System.IO.File	Description
`Open: (path : string)* (mode : FileMode) -> FileStream`	Request the opening of a file on `path` for reading and writing with access mode `FileMode`, see Table 12.2. Other programs are not allowed to access the file, before this program closes it.
`OpenRead: (path : string) -> FileStream`	Request the opening of a file on `path` for reading. Other programs may read the file regardless of this opening.
`OpenText: (path : string) -> StreamReader`	Request the opening of an existing UTF8 file on `path` for reading. Other programs may read the file regardless of this opening.
`OpenWrite: (path : string) -> FileStream`	Request the opening of a file on `path` for writing with `FileMode.OpenOrCreate`. Other programs may not access the file, before this program closes it.
`Create: (path : string) -> FileStream`	Request the creation of a file on `path` for reading and writing, overwriting any existing file. Other programs may not access the file, before this program closes it.
`CreateText: (path : string) -> StreamWriter`	Request the creation of an UTF8 file on `path` for reading and writing, overwriting any existing file. Other programs may not access the file, before this program closes it.

Table 12.1: The family of `System.IO.File.Open` functions. See Table 12.2, 12.3, 12.4, 12.5, 12.6, 12.7, and 12.8 for the description of `FileMode`, `FileStream`, `StreamWriter`, and `StreamReader`.

12.2 Storing and retrieving data from a file

A file stored on the filesystem has a name, and it must be opened before it can be accessed and closed when finished. Opening files informs the operating system that your program is now going to use the file, and your program may request protection of the file from the operating system. E.g., if you are going to write to the file, then this typically implies that no one else may write to the file at the same time. Thus we typically say, that you reserve a file by opening it, and you release it again by closing it, such that other programs may have access to it. On the other hand, it is typically safe for several programs to read the same file at the same time, but it is still important to close files after their use, such that the operating system can effectively manage the computer's resources. Conversely, you may not succeed in opening a file, since it may not exist, you may not have sufficient rights for accessing it, or other programs may at the moment have reserved it for their use. Thus, **never assume that accessing files always works, but program defensively, e.g., by checking the return status of the file accessing functions and by `try` constructions.** Advice

Data in a file may have been stored in files in various ways, e.g., it may contain UTF8 encoded characters or sequences of floating point numbers stored as raw bits in chunks of 64 bits, or it may be a sequence of bytes that are later going to be interpreted as an image in jpeg or tiff format. To aid in retrieving the data, F# has a family of open functions, all residing in the `System.IO.File` class. These are described in Table 12.1. For the general `Open` function, you must also specify how the file is to be opened. This is done with a special set of values described in Table 12.2. An example of how a file is opened and later closed is,

FileMode.	Description
Append	Open a file and seek to its end, if it exists, or create a new file. Can only be used together with FileAccess.Write. May throw IOException and NotSupportedException exceptions.
Create	Create a new file, and delete an already existing file. May throw the UnauthorizedAccessException exception.
CreateNew	Create a new file, but throw the IOException exception, if the file already exists.
Open	Open an existing file, and System.IO.FileNotFoundException exception is thrown if the file does not exist.
OpenOrCreate	Open a file, if exists, or create a new file.
Truncate	Open an existing file and truncate its length to zero. Cannot be used together with FileAccess.Read.

Table 12.2: File mode values for the System.IO.Open function.

Property	Description
CanRead	Gets a value indicating whether the current stream supports reading.(Overrides Stream.CanRead.)
CanSeek	Gets a value indicating whether the current stream supports seeking.(Overrides Stream.CanSeek.)
CanWrite	Gets a value indicating whether the current stream supports writing.(Overrides Stream.CanWrite.)
Length	Gets the length in bytes of the stream.(Overrides Stream.Length.)
Name	Gets the name of the FileStream that was passed to the constructor.
Position	Gets or sets the current position of this stream.(Overrides Stream.Position.)

Table 12.3: Some properties of the System.IO.FileStream class.

Listing 12.2, openFile.fsx:
Opening and closing a file, in this case the source code of this same file.

```
let filename = "openFile.fsx"
// Open the file and return the stream value as an option type
let reader =
  try
    Some (System.IO.File.Open (filename, System.IO.FileMode.Open))
  with
    _ -> None

// Do something with the file
if reader.IsSome then
  printfn "The file %A was successfully opened." filename

// If the file was opened, then it must be closed
if reader.IsSome then
  reader.Value.Close ()
```

```
The file "openFile.fsx" was successfully opened.
```

The return value from the open family of commands is a System.IO.FileStream class. It has a number of members and methods that allow you to read and write to the file and to obtain further information about the file. Some important properties and methods are stated in Table 12.3 and 12.4.

Method	Description
`Close ()`	Closes the stream.
`Flush ()`	Causes any buffered data to be written to the file.
`Read byte[] * int * int`	Reads a block of bytes from the stream and writes the data in a given buffer.
`ReadByte ()`	Read a byte from the file and advances the read position to the next byte.
`Seek int * SeekOrigin`	Sets the current position of this stream to the given value.
`Write byte[] * int * int`	Writes a block of bytes to the file stream.
`WriteByte byte`	Writes a byte to the current position in the file stream.

Table 12.4: Some methods of the `System.IO.FileStream` class.

Property	Description
`EndOfStream`	Check whether the stream is at its end.

Table 12.5: a property of the `System.IO.StreamReader` class.

Method	Description
`Close ()`	Closes the stream.
`Flush ()`	Causes any buffered data to be written to the file.
`Peek ()`	Reads the next character, but does not advance the position.
`Read ()`	Reads the next character.
`Read char[] * int * int`	Reads a block of bytes from the stream and writes the data in a given buffer.
`ReadLine ()`	Reads the next line of characters until a newline. Newline is discarded.
`ReadToEnd ()`	Reads the remaining characters till end-of-file.

Table 12.6: Some methods of the `System.IO.StreamReader` class.

Property	Description
`AutoFlush : bool`	Get or set the auto-flush. If set, then every call to `Write` will flush the stream.

Table 12.7: a property of the `System.IO.StreamWriter` class.

Method	Description
`Close ()`	Closes the stream.
`Flush ()`	Causes any buffered data to be written to the file.
`Write 'a`	Write a basic type to the file.
`WriteLine string`	As `Write` but followed by newline.

Table 12.8: Some methods of the `System.IO.StreamWriter` class.

Function	Description
`Copy (src : string, dest : string)`	Copy a file from `src` to `dest` possibly overwriting any existing file.
`Delete string`	Delete a file
`Exists string`	Check whether the file exists
`Move (from : string, to : string)`	Move a file from `src` to `to` possibly overwriting any existing file.

Table 12.9: Some methods of the `System.IO.File` class.

A simple example of opening a text-file and processing it is,

Listing 12.3, readFile.fsx:
An example of opening a text file, and using the `StreamReader` properties and methods.

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

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

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

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

Here the program reads the source code of itself, and prints it to the console.

12.3 Working with files and directories.

[3] In the `System.IO.File` class there are a number of other frequently used functions summarized in Table 12.9

In the `System.IO.Directory` class there are a number of other frequently used functions summarized in Table 12.10

In the `System.IO.Path` class there are a number of other frequently used functions summarized in Table 12.11

[3]Todo: See `https://msdn.microsoft.com/en-us/library/ms404278(v=vs.110).aspx`

Function	Description
`CreateDirectory string`	Create the directory and all implied sub-directories.
`Delete string`	Delete a directory
`Exists string`	Check whether the directory exists
`GetCurrentDirectory ()`	Get working directory of the program
`GetDirectories (path : string)`	Get directories in `path`
`GetFiles (path : string)`	Get files in `path`
`Move (from : string, to : string)`	Move a directory and its content from `src` to `to`.

Table 12.10: Some methods of the `System.IO.Directory` class.

Function	Description
`Combine string * string`	Combine 2 paths into a new path.
`GetDirectoryName (path: string)`	Extract the directory name from `path`.
`GetExtension (path: string)`	Extract the extension from `path`.
`GetFileName (path: string)`	Extract the name and extension from `path`.
`GetFileNameWithoutExtension (path : string)`	Extract the name without the extension from `path`.
`GetFullPath (path : string)`	Extract the absolute path from `path`.
`GetTempFileName ()`	Create a uniquely named and empty file on disk and return its full path.

Table 12.11: Some methods of the `System.IO.Path` class.

12.4 Programming intermezzo

A typical problem, when working with files, is

Problem 12.1:

Ask the user for the name of an existing file.

Such a dialogue most often requires the program to aid the user, e.g., by telling the user, which files are available, and to check that the filename entered is an existing file. A solution could be,

Listing 12.4:

```
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
```

A practice problem could be,

Problem 12.2:
Ask the user for the name of an existing file, read the file and print it in reverse order.

This could be solved as,

Listing 12.5, reverseFile.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"]
```

Part II

Imperative programming

[4]

[4] Todo: **Remember, `do` expr, where `do` is optional and expr must return unit.**

Chapter 13

Graphical User Interfaces

...

Chapter 14

Imperative programming

14.1 Introduction

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

· Imperativ programming
· statements
· states

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.

· Procedural programming
· Object oriented programming

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 langues follow the imperative programming paradigm.

· machine code

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.

· Functional programming

14.2 Generating random texts

14.2.1 0'th order statistics

Listing 14.1, randomTextOrder0.fsx:

```
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
```

14.2.2 1'th order statistics

Listing 14.2, randomTextOrder1.fsx:

```
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
```

Part III

Declarative programming

Chapter 15

Sequences and computation expressions

[1]

15.1 Sequences

Sequences are lists, where the elements are build as needed. Examples are[2]

Listing 15.1, seqExample.fsx:
Creating sequences by range explicitly stating elements, a range expressions, a computation expression, and an infinite computation expression

```
> #nowarn "40"
- let a = { 1 .. 10 };;

val a : seq<int>

> let b = seq { 1 .. 10 };;

val b : seq<int>

> let c = seq {for i = 1 to 10 do yield i*i done};;

val c : seq<int>

> let rec d =
-   seq {
-     for i = 0 to 59 do yield (float i)*2.0*3.1415/60.0 done;
-     yield! d
-     };;

val d : seq<float>
```

[1]Todo: **possibly add maps and sets as well.**
[2]Todo: **Mono does not support specification Spec-4.0 Section 6.3.11, seq comp-expr, in the form seq 3 or seq 3; 4.**

Sequences are built using the following subset of the general syntax,

```
range-exp = expr ".." expr [".." expr]
comp-expr =
  "let" pat "=" expr "in" comp-expr
  | "use" pat = expr "in" comp-expr
  | ("yield" | "yield!") 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
short-comp-expr = "for" pat "in" (expr | range-expr) "->" expr
comp-or-range-expr = comp-expr| short-comp-expr | range-expr
comp-rule = pat pattern-guardopt "->" comp-expr
comp-rules = comp-rule | comp-rule '|' comp-rules
expr = ...
  | "seq" "{" comp-or-range-expr "}" (* computation expression *)
  | ...
```

Sequence may be defined using simple range expressions but most often are defined as a small program, that generates values with the `yield` keyword or `yield!` keyword. The `yield!` is called *yield bang*, and appends a sequence instead of adding a sequence as an element. Thus, `seq {3; 5}` is not permitted, but `seq {yield 3; yield 5}` and `seq {yield!(seq {yield 3; yield 5})}` are, both creating `seq <int> = seq [3; 5]`, i.e., a sequence of integers. Most often computation expressions are used to produced members that are not just ranges, but more complicated expressions of ranges, e.g., `c` in the example. Sequences may even in principle be infinitely long, e.g., `d`. Calculating the complete sequence at the point of definition is impossible due to lack of memory, as is accessing all its elements due to lack of time. But infinite sequences are still very useful, e.g., identifier `d` illustrates the parametrization of a circle, which is an infinite domain, and any index will be converted to the equivalent 60th degree angle in radians. F# warns against recursive values, as defined in the example, since it will check the soundness of the value at run-time rather than at compile-time. The warning can be removed by adding `#nowarn "40"` in the script or `--nowarn:40` as argument to `fsharpi` or `fsharpc`. · yield bang

Sequences are generalisations of lists and arrays, and functions taking sequences as argument may equally take lists and arrays as argument. Sequences do not have many built-in operators, but a rich collection of functions in the `Collections.Seq`. E.g.,

Listing 15.2, seqIndexing.fsx:
Index a sequence with `Seq.item` and `Seq.take`

```
> let sq = seq { 1 .. 10 };; (* make a sequence *)

val sq : seq<int>

> let itm = Seq.item 0 sq;; (* take firste element *)

val itm : int = 1

> let sbsq = Seq.take 3 sq;; (* make new sequence of first 3 elements *)

val sbsq : seq<int>
```

which as usual index from 0 and will cast an exception, if indexing is out of range for the sequence.

That sequences really are programs rather than values can be seen by the following example,

Listing 15.3, seqDelayedEval.fsx:
Sequences elements are first evaluated, when needed.

```
1
That was 0
2
That was 1
The sequence was evaluated to this point.
3
That was 2
```

In the example, we see that the `printfn` function embedded in the definition is first executed, when the 3rd item is requested.

[3]

The only difference between computation expression's programming constructs and the similar regular expressions constructs is that they must return a value with the `yield` or `yield!` keywords.[4] The `try`-keyword constructions will be discussed in Chapter 11, and the `use`-keyword is a variant of `let` but used in asynchronous computations, which will not be treated here.

Infinite sequences is a useful concept in many programs and may be generated in a number of ways. E.g., to generate a repeated sequence, we could use recursive value definition, a computation expression, a recursive function, or the `Seq` module. Using a recursive value definition,

Listing 15.4, seqInfinteValue2.fsx:
Recursive value definitions gives a warning. Compare with Listing15.5, 15.6, and 15.7.

```
let repeat items =
  let rec ret =
    seq { yield! items
          yield! ret }
  ret

printfn "%A" (repeat [1;2;3])
```

```


/Users/sporring/repositories/fsharpNotes/src/seqInfinteValue2.fsx(4,18):
    warning FS0040: This and other recursive references to the object(s)
    being defined will be checked for initialization-soundness at runtime
    through the use of a delayed reference. This is because you are
    defining one or more recursive objects, rather than recursive
    functions. This warning may be suppressed by using '#nowarn "40"' or
    '--nowarn:40'.
seq [1; 2; 3; 1; ...]
```

F# warns against using recursive values, since it will check the soundness of the value at run-time

[3]Todo: **Mono, missing support for Spec-4.0 Chapter 6, `do-in` in sequences. E.g., `seq let _ = printfn "hej "in yield 3` is ok, but `seq do printfn "hej"in yield 3` not. One could argue, that computation expression is the framework and that it is the seq implementation, which does not provide full access to the framework. but this is confusing, since seq gets special attention in the specification.**

[4]Todo: **Mono implements if-elif-else, but this is not in the specification.**

rather than at compile-time. The warning can be removed by adding `#nowarn "40"` in the script or `--nowarn:40` as argument to `fsharpi` or `fsharpc`, but **warnings are messages from the designers of F# that your program is non-optimal, and you should avoid structures that throw warnings instead of relying on `#nowarn` and similar constructions.** Instead we may create an infinite loop using the `while-do` computation expression, as Advice

Listing 15.5, seqInfinteValue.fsx:
Infinite value definition without recursion nor warning. Compare with Listing 15.4, 15.6, and15.7.

```
let repeat items =
  seq { while true do yield! items }

printfn "%A" (repeat [1;2;3])
```

```
seq [1; 2; 3; 1; ...]
```

or alternatively define a recursive function,

Listing 15.6, seqInfinteFunction.fsx:
Recursive function definitions gives no a warning. Compare with Listing 15.4, 15.5, and 15.7.

```
let rec repeat items =
  seq { yield! items
        yield! repeat items }

printfn "%A" (repeat [1;2;3])
```

```
seq [1; 2; 3; 1; ...]
```

Infinite expressions have built-in support through the `Seq` module using ,

Listing 15.7, seqInitInfinite.fsx:
Using `Seq.initInfinte` and a function to generate an infinte sequence. Compare with Listing 15.4, 15.5, and 15.6.

```
let repeat items =
  let get items x = Seq.item (x % (Seq. length items)) items
  Seq.initInfinite (get items)

printfn "%A"  (repeat [1;2;3])
```

```
seq [1; 2; 3; 1; ...]
```

which takes a function as argument. Here we have used currying, i.e., `get items` is a function that takes on variable and returns a value. The use of the remainder operator makes the example rather contrived, since it might have been simpler to use the `get` indexing function directly.

Sequences are easily converted to and from lists and arrays as,

Listing 15.8, seqConversion.fsx:
Conversion between sequences and lists and arrays using the `List` module.

```
let sq = seq { 1 .. 3 }
let lst = Seq.toList sq (* convert sequence to list *)
let arr = Seq.toArray sq (* convert sequence to array *)
let sqFromArr = seq [| 1 .. 3|] (* convert an array to sequence *)
let sqFromLst = seq [1 .. 3] (* convert a list to sequence *)
printfn "%A, %A, %A, %A, %A" sq lst arr sqFromArr sqFromLst
```

```
seq [1; 2; 3], [1; 2; 3], [|1; 2; 3|], [|1; 2; 3|], [1; 2; 3]
```

There are quite a number of built-in functions for sequences many which will be discussed in Chapter F.

Lists and arrays may be created from sequences through the short-hand notation called *list and array sequence expressions*, · list sequence expression

```
expr = ...
  | "[" (... | comp-expr | short-comp-expr | ...) "]" (* list sequence expression
      *)
  | "[|" (.. | comp-expr | short-comp-expr | ...) "|]" (* array sequence
      expression *)
  | ...
```

which implicitly creates the corresponding expression and return the result as a list or array.

Chapter 16

Patterns

16.1 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

Problem 16.1:

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,

Listing 16.1, matchWith.fsx:
Using the match-keywordwith programming construct to vary calculation based on the input value.

```
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"
```

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 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 \lstinline|` _ -> "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. Match with can only take one identifier, but this can be tuples for matching with combinations of identifiers, see Chapter 9 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

Listing 16.2, functionKeyword.fsx:
Function definition and match expressions are integrated using the function keyword. Compare with Listing 16.1

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

Comparing with Listing 16.1 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.**[1] Advice

[1] Todo: **Should we extend this with a more detail description of possiblities from Spec-4.0 Chapter 7?**

Chapter 17

Types and measures

17.1 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,

Listing 17.1: fsharpi, floating point and integer numbers may be assigned unit of measures.

```
> [<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
```

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

Listing 17.2: fsharpi, unit of measures adds an extra layer of types for syntax checking at compile time.

```
> [<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'
```

Unit of measures allow for *, /, and ^[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

Listing 17.3: fsharpi, typecasting unit of measures.

```
> [<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
```

Compound symbols can be declared as,

Listing 17.4: fsharpi, aggregated unit of measures.

```
> [<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
```

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 17.1. Hence, using the predefined unit of seconds, we may write,

Listing 17.5: fsharpi, SI unit of measures are built-in.

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

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

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

Listing 17.6: fsharpi, simpler syntax by importing, but beware of namespace pollution.

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

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

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.

[1]Todo: **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 17.1: International System of Units.

We have obtained, what is known as *namespace pollution*. Read more about namespaces in Part IV.

· namespace pollution

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.

Chapter 18

Functional programming

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

Listing 18.1, listPatternMatching.fsx:
Examples of list concatenation, indexing.

```
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
```

The pattern `l::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

Problem 18.1:

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 18.1 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,

Listing 18.2, avgGradesRec.fsx:
Calculating a list of average grades using recursion and pattern matching.

```
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
```

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,

Listing 18.3, listSquare.fsx:
Using pattern matching and list appending elements to lists.

```
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]
```

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]` $\curvearrowright$ `1 * 1 :: square [2 .. 10]` $\curvearrowright$ `1 * 1 :: (2 * 2 :: square [3 .. 10])` $\curvearrowright \ldots$ `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 18

Arrays only support direct pattern matching, e.g.,

Listing 18.4, arrayPatternMatching.fsx:
Only simple pattern matching is allowed for arrays.

```
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
```

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`. Below further discussion of on 2 and higher dimensional arrays be discussed.

Part IV

Structured programming

Chapter 19

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
  | anonynmous-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 multipe namespaces each defining types and modules, these in turn may contain function and value definitions, which in turn contains expressions.[1]

[1] Todo: **Spec-4.0 Chapter 10.**

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.[2]

Namespaces is an optional hierarchial catergorization 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.[3]

[2]Todo: `https://en.wikibooks.org/wiki/F_Sharp_Programming/Modules_and_Namespaces`
[3]Todo: `https://fsharpforfunandprofit.com/posts/organizing-functions/`

Chapter 20

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 $d \in \{0, 1, 2, \ldots, 9\}$ and the weight of each digit is proportional to its place in the sequence of digits w.r.t. the decimal point, i.e., the number $357.6 = 3 \cdot 10^2 + 5 \cdot 10^1 + 7 \cdot 10^0 + 6 \cdot 10^{-1}$ or in general:

· decimal number
· base
· decimal point
· digit

$$v = \sum_{i=-m}^{n} d_i 10^i \tag{A.1}$$

The basic unit of information in almost all computers is the binary digit or *bit* for short. A *binary* number consists of a sequence of binary digits separated by a decimal point, where each digit can have values $b \in \{0, 1\}$, and the base is 2. The general equation is,

· bit
· binary

$$v = \sum_{i=-m}^{n} b_i 2^i \tag{A.2}$$

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*.

· most significant bit
· least significant bit
· byte
· word

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

· octal
· hexadecimal

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

· reals

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.

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,

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

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

where s, e_i, and m_j are binary digits. The bits are converted to a number using the equation by first calculating the exponent e and the mantissa m,

$$e = \sum_{i=1}^{11} e_i 2^{11-i}, \tag{A.3}$$

$$m = \sum_{j=1}^{52} m_j 2^{-j}. \tag{A.4}$$

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,

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

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)
$e = 2^{11} - 1$	$v = (-1)^s\, \infty$	$v = (-1)^s$ NaN (not a number)

· subnormals
· NaN
· not a number

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

$$e = 2^{11} - 2 = 2046 \tag{A.6}$$

$$m = \sum_{j=1}^{52} 2^{-j} = 1 - 2^{-52} \simeq 1. \tag{A.7}$$

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

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

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

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

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

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

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

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

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,

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

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

$$\begin{aligned} v &= (-1)^s \left(\sum_{j=1}^{52} m_j 2^{-j} \right) 2^{-1022} && \text{(A.15)} \\ &= \pm \left(\sum_{j=1}^{52} m_j 2^{-j} 2^{-1022} \right) && \text{(A.16)} \\ &= \pm \left(\sum_{j=1}^{52} m_j 2^{-j-1022} \right) && \text{(A.17)} \\ &\overset{k=-j-1022}{=} \pm \left(\sum_{j=-1023}^{-1074} m_{-k-1022} 2^k \right) && \text{(A.18)} \end{aligned}$$

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 comunicate 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.

· American Standard Code for Information Interchange
· ASCII

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

· ASCIIbetical order

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

Table B.1: ASCII

B.2 ISO/IEC 8859

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

· Latin1

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.

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

A unicode code point is an abstraction from the encoding and the graphical representation of a character. A code point is written as“U+” followed by its hexadecimal number, and for the Basic Multilingual plane 4 digits are used, e.g., the code point with the unique name LATIN CAPITAL LETTER A has the unicode code point is “U+0041”, and is in this text it is visualized as ’A’. More digits are used for code points of the remaining planes.

The general category is used in grammars to specify valid characters, e.g., in naming identifiers in F#.

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 hypen

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

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.

· UTF-8

· UTF-16

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 Nauer for his work on ALGOL 60.

· Extended Backus-Naur Form
· EBNF

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

· terminal symbols
· production rules

```
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 { 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`.

'{ ... }' 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.

```
codepoint = ?Any unicode codepoint?;
```

'`-`' 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.

Rules for rewriting EBNF are:

<table>
<tr><th>Rule</th><th>Description</th></tr>
<tr><td><code>s | t</code> ↔ <code>t | s</code></td><td>| is commutative</td></tr>
<tr><td><code>r | (s | t)</code> ↔ <code>(r | s)| t</code> ↔ <code>r | s | t</code></td><td>| is associative</td></tr>
<tr><td><code>(r s)t</code> ↔ <code>r (s t)</code> ↔ <code>r s t</code></td><td>concatenation is associative</td></tr>
<tr><td><code>r (s | t)</code> ↔ <code>r t| r s</code></td><td>concatenation is distributive over |</td></tr>
<tr><td><code>(r | s)t</code> ↔ <code>r t| r t</code></td><td></td></tr>
<tr><td><code>[s | t]</code> ↔ <code>[t] | [s]</code></td><td></td></tr>
<tr><td><code>[[s]]</code> ↔ <code>[s]</code></td><td><code>[]</code> is idempotent</td></tr>
<tr><td><code>{{s}}</code> ↔ <code>{s}</code></td><td><code>{}</code> is idempotent</td></tr>
</table>

where `r`, `s`, and `t` are production rules or terminals. Precedence for the EBNF symbols are,

<table>
<tr><th>Symbol</th><th>Description</th></tr>
<tr><td>*</td><td>repetition</td></tr>
<tr><td>-</td><td>except</td></tr>
<tr><td>,</td><td>concatenate</td></tr>
<tr><td>|</td><td>option</td></tr>
<tr><td>=</td><td>define</td></tr>
<tr><td>;</td><td>terminator</td></tr>
</table>

in order of precedence, such that $\star$ has higher precedence than $-$. These precedence rules are overridden by bracket pairs, such as `' '`, `" "`, `(* *)`, `( )`, `[ ]`, `{ }`, `? ?`.

The proposal allows for identifies that includes space, but often a reduced form is used, where identifiers are single words, in which case the concatenation symbol `,` is replaced by a space. 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 = "_";
space = " ";
newline = ?a newline character?;
identifier = letter  { letter | digit | underscore };
character = letter | digit | symbol | underscore;
string = character  { character };
terminal = "'"  string  "'" | '"'  string  '"';
rhs = identifier
  | terminal
  | "["  rhs  "]"
  | "{"  rhs  "}"
  | "("  rhs  ")"
  | "?"  string  "?"
  | rhs  "|"  rhs
  | rhs  ","  rhs
  | rhs  space  rhs; (*relaxed ebnf*)
rule = identifier  "="  rhs ";"
  | identifier  "="  rhs newline; (*relaxed ebnf*)
grammar = rule { rule };
```

Here the comments demonstrate, the relaxed modification. Newline does not have an explicit representation in EBNF, which is why we use `? ?` brackets

Appendix D

F♭

Minimal F# used in Part I

Listing D.1: F♭, a subset of F#

```
(*Special characters*)
codePoint = ?Any unicode codepoint?;
Lu = ?Upper case letters?;
Ll = ?Lower case letters?;
Lt = ?Digraphic letters, with first part uppercase?;
Lm = ?Modifier letters?;
Lo = ?Gender ordinal indicators?;
Nl = ?Letterlike numeric characters?;
Pc = ?Low lines?;
Mn = ?Nonspacing combining marks?;
Mc = ?Spacing combining marks?;
Cf = ?Soft Hyphens?;

(*Whitespace*)
whitespace = " " {" "};
newline = "\n" | "\r" "\n";

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

(*Literal digits*)
dDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
bDigit = "0" | "1";
oDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7";
xDigit =
  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
  | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" | "f";

(*Literal integers*)
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;
```

```
sbyte = (dInt | xInt) "y";
byte = (dInt | xInt) "uy";
int32 = (dInt | xInt) ["l"];
uint32 = (dInt | xInt) ("u" | "ul");

(*Literal floats*)
float = dFloat | sFloat;
dFloat = dInt "." {dDigit};
sFloat = (dInt | dFloat) ("e" | "E" ) ["+" | "-"] dInt;
ieee64 = float | xInt "LF";

(*Literal chars*)
char = "'" codePoint | escapeChar "'";
escapeChar =
  "\" ("b" | "n" | "r" | "t" | "\" | '"' | "'" | "a" | "f" | "v")
  | "\u" xDigit xDigit xDigit xDigit
  | "\U" xDigit xDigit xDigit xDigit xDigit xDigit xDigit xDigit
  | "\" dDigit dDigit dDigit;

(*Literal strings*)
string = '"' { stringChar }  '"';
stringChar = char - '"';
verbatimString = '@"' {char - ('"' | '\"' )| '""'} '"';

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

(*Expressions*)
expr =
  const (*a const value*)
  | "(" expr ")" (*block*)
  | longIdentOrOp (*identifier or operator*)
  | expr "." longIdentOrOp (*dot lookup expression, no space around "."*)
  | expr expr (*application*)
  | expr infixOp expr (*infix application*)
  | prefixOp expr (*prefix application*)
  | expr ".[" expr "]" (*index lookup, no space before "."*)
  | expr ".[" sliceRange "]" (*index lookup, no space before "."*)
```

```
  | expr "<-" expr (*assingment*)
  | exprTuple (*tuple*)
  | "[" (exprSeq | rangeExpr) "]" (*list*)
  | "[|" (exprSeq | rangeExpr) "|]" (*array*)
  | expr ":" type (*type annotation*)
  | expr ";" expr (*sequence of expressions*)
  | "let" valueDefn "in" expr (*binding a value or variable*)
  | "let" ["rec"] functionDefn "in" expr (*binding a function or operator*)
  | "fun" argumentPats "->" expr (*anonymous function*)
  | "if" expr "then" expr {"elif" expr "then" expr} ["else" expr] (*conditional*)
  | "while" expr "do" expr ["done"] (*while*)
  | "for" ident "=" expr "to" expr "do" expr ["done"] (* simple for expression *)
  | "try" expr "with"  ["|"] rules (*exception*)
  | "try" expr "finally" expr; (*exception with cleanup*)
exprTuple = expr | expr "," exprTuple;
exprSeq =  expr | expr ";" exprSeq;
rangeExpr = expr ".." expr [".." expr];
sliceRange =
  expr
  | expr ".." (*no space between expr and ".."*)
  | ".." expr (*no space between expr and ".."*)
  | expr ".." expr (*no space between expr and ".."*)
  | "*";

(*Constants*)
const =
  byte
  | sbyte
  | int32
  | uint32
  | int
  | ieee64
  | char
  | string
  | verbatimString
  | "false"
  | "true"
  | "()";

(*Identifiers*)
ident = (letter | "_") {letter | dDigit | specialChar};
letter = Lu | Ll | Lt | Lm | Lo | Nl; (*e.g. "A", "B" ... and "a", "b", ...*)
specialChar = Pc | Mn | Mc | Cf; (*e.g., "_"*)

longIdent = ident | ident "." longIdent; (*no space around "."*)
longIdentOrOp = [longIdent "."] identOrOp; (*no space around "."*)
identOrOp =
  ident
  | "(" infixOp | prefixOp ")"
  | "(*)";

(*Keywords*)
identKeyword =
  "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" | "yield";

reservedIdentKeyword =
  "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" | "volatile";

reservedIdentFormats = ident ( "!" | "#");

(*Symbolic Keywords*)
symbolicKeyword =
  "let!" | "use!" | "do!" | "yield!" | "return!" | "|" | "->" | "<-" | "." | ":"
  | "(" | ")" | "[" | "]" | "[<" | ">]" | "[|" | "|]" | "{" | "}" | "'" | "#"
  | ":?>" | ":?" | ":>" | ".." | "::" | ":=" | ";;" | ";" | "=" | "_" | "?"
  | "??" | "(*)" | "<@" | "@>" | "<@@" | "@@>";

reservedSymbolicSequence =  "~" | "'";

(*Types*)
type =
  longIdent (*named such as "int"*)
  | "(" type ")" (*paranthesized*)
  | type "->" type (*function*)
  | typeTuple (*tuple*)
  | "'" ident (*variable, no space after "'"*)
  | type longIdent (*named such as "int list"*)
  | type "[" typeArray "]"; (*array, no spaces*)
typeTuple = type | type "*" typeTuple;
typeArray = "," | "," typeArray;

(*Value definition*)
valueDefn = ["mutable"] pat "=" expr;

(*Patterns*)
pat =
  const (*constant*)
  | "_" (*wildcard*)
  | ident (*named*)
  | pat "::" pat (*construction*)
  | pat ":" type (*type constraint*)
  | "(" pat ")" (*paranthesized*)
  | patTuple (*tuple*)
  | patList (*list*)
  | patArray (*array*)
  | ":?" type; (*dynamic type test*)
patTuple = pat | pat "," patTuple;
patList = "[" [patSeq] "]";
patArray = "[|" [patSeq] "|]";
patSeq = pat | pat ";" patSeq;

(*Function definition*)
functionDefn = identOrOp argumentPats [":" type] "=" expr;
argumentPats = pat | pat argumentPats;

(*Rules*)
```

```
rules = rule | rule "|" rules;
rule = pat ["when" expr] "->" expr;

(*script-file*)
moduleElems = moduleElem | moduleElem moduleElems;
moduleElem =
  "let" valueDefn "in" expr (*binding a value or variable*)
  | "let" ["rec"] functionDefn "in" expr (*binding a function or operator*)
  | "exception" ident of typeTuple (*exception definition*)
  | "open" longIdent (*import declaration*)
  | "#" ident string; (*compiler directive, no space after "#"*)
```

[1]

[1] Todo: **I don't think we need `type="'"ident` nor `moduleelm = "#"ident string`**

Appendix E

Language Details

This appendix lists various language details.

E.1 Arithmetic operators on basic types

Operator	leftOp	rightOp	Expression	Result	Description
leftOp + rightOp	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
leftOp - rightOp	ints	ints	5 - 2	3	Subtraction
	floats	floats	5.0 - 2.0	3.0	
leftOp * rightOp	ints	ints	5 * 2	10	Multiplication
	floats	floats	5.0 * 2.0	10.0	
leftOp / rightOp	ints	ints	5 / 2	2	Integer division
	floats	floats	5.0 / 2.0	2.5	Division
leftOp % rightOp	ints	ints	5 % 2	1	Remainder
	floats	floats	5.0 % 2.0	1.0	
leftOp ** rightOp	floats	floats	5.0 ** 2.0	25.0	Exponentiation
leftOp && rightOp	bool	bool	true && false	false	boolean and
leftOp || rightOp	bool	bool	true || false	false	boolean or
leftOp &&& rightOp	ints	ints	0b1010 &&& 0b1100	0b1000	bitwise bool and
leftOp ||| rightOp	ints	ints	0b1010 ||| 0b1100	0b1110	bitwise boolean or
leftOp ^^^ rightOp	ints	ints	0b1010 ^^^ 0b1101	0b0111	bitwise boolean exclusive or
leftOp <<< rightOp	ints	ints	0b00001100uy <<< 2	0b00110000uy	bitwise shift left
leftOp >>> rightOp	ints	ints	0b00001100uy >>> 2	0b00000011uy	bitwise and
+op	ints		+3	3	identity
	floats		+3.0	3.0	
-op	ints		-3	-3	negation
	floats		-3.0	-3.0	
not op	bool		not true	false	boolean negation
~~op	ints		~~~0b00001100uy	0b11110011uy	bitwise boolean negation

Table E.1: 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.

Operator	`leftOp`	`rightOp`	Expression	Result	Description
`leftOp < rightOp`	bool	bool	`true < false`	`false`	Less than
	ints	ints	`5 < 2`	`false`	
	floats	floats	`5.0 < 2.0`	`false`	
	chars	chars	`'a' < 'b'`	`true`	
	strings	strings	`"ab" < "cd"`	`true`	
`leftOp > rightOp`	bool	bool	`true > false`	`true`	Greater than
	ints	ints	`5 > 2`	`true`	
	floats	floats	`5.0 > 2.0`	`true`	
	chars	chars	`'a' > 'b'`	`false`	
	strings	strings	`"ab" > "cd"`	`false`	
`leftOp = rightOp`	bool	bool	`true = false`	`false`	Equal
	ints	ints	`5 = 2`	`false`	
	floats	floats	`5.0 = 2.0`	`false`	
	chars	chars	`'a' = 'b'`	`false`	
	strings	strings	`"ab" = "cd"`	`false`	
`leftOp <= rightOp`	bool	bool	`true <= false`	`false`	Less than or equal
	ints	ints	`5 <= 2`	`false`	
	floats	floats	`5.0 <= 2.0`	`false`	
	chars	chars	`'a' <= 'b'`	`true`	
	strings	strings	`"ab" <= "cd"`	`true`	
`leftOp >= rightOp`	bool	bool	`true >= false`	`true`	Greater than or equal
	ints	ints	`5 >= 2`	`true`	
	floats	floats	`5.0 >= 2.0`	`true`	
	chars	chars	`'a' >= 'b'`	`false`	
	strings	strings	`"ab" >= "cd"`	`false`	
`leftOp <> rightOp`	bool	bool	`true <> false`	`true`	Not Equal
	ints	ints	`5 <> 2`	`true`	
	floats	floats	`5.0 <> 2.0`	`true`	
	chars	chars	`'a' <> 'b'`	`true`	
	strings	strings	`"ab" <> "cd"`	`true`	

Table E.2: Comparison operators on basic types. Ints, floats, chars, and strings means all built-in integer types etc..

E.2 Basic arithmetic functions

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

Table E.3: Predefined functions for arithmetic operations

Name	Example	Description
`fst`	`fst (1, 2)`	
`snd`	`snd (1, 2)`	
`failwith`	`failwith`	
`invalidArg`	`invalidArg`	
`raise`	`raise`	
`reraise`	`reraise`	
`ref`	`ref`	
`ceil`	`ceil`	

Table E.4: Built-in functions.

E.3 Precedence and associativity

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

Table E.5: Some common operators, their precedence, and their associativity. Rows are ordered from highest to lowest precedences, such that `leftOp * rightOp` has higher precedence than `leftOp + rightOp`. Operators in the same row has same precedence. Full table is given in Table E.6.

· boolean or
· boolean and

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

Table E.6: Precedence and associativity of operators. Operators in the same row has same precedence. See Listing 6.3 for the definition of `prefixOp`

E.4 Lightweight Syntax

To appear later.[1]

[1] Todo: **See Lightweight Syntax, Spec-4.0 Chapter 15.1**

Appendix F

The Some Basic Libraries

[1]

[2] [3]

F.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.

[1] Todo: **Work in progress!**

[2] Todo: **Discuss `Fsharp.Core` and `System` and all the operators and functions defined there.**

[3] Todo: **See** `https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/import-declarations-the-open-keyword-fsharp` **for namespaces opened per default.**

`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.

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 x for each element for which the function returns Some(x).
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 System.Collections.Generic.KeyNotFoundException if no such element exists.
findIndex	Returns the index of the first element in an array that satisfies the supplied condition. Raises System.Collections.Generic.KeyNotFoundException if none of the elements satisfy the condition.
fold	Applies a function to each element of an array, threading an accumulator argument through the computation. If the input function is f and the array elements are i0...iN, this function computes f (...(f s i0)...) iN.
foldBack	Applies a function to each element of an array, threading an accumulator argument through the computation. If the input function is f and the array elements are i0...iN, this function computes f i0 (...(f iN s)).
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 System.Array.Length 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. Operators.max is used to compare the elements.
min	Returns the smallest of all elements of an array. Operators.min 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. Operators.compare is used to compare the elements.
sub	Creates an array that contains the sup<plied 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, System.ArgumentException is raised.

Table F.1: Some built-in procedures in the Array module for arrays (from `https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/fsharp-core-library-reference`)

F.2 List, arrays, and sequences

In Table F.1. Thus, the `arrayReassign.fsx` program can be written using arrays as,

Listing F.1, arrayReassignModule.fsx:

```
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
```

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

Listing F.2, flowForListsIndexModule.fsx:

```
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
```

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 F.2

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 F.2: Some built-in procedures in the Array2D module for arrays (from `https://msdn.microsoft.com/en-us/visualfsharpdocs/conceptual/fsharp-core-library-reference`)

F.3 Mutable Collections

`System.Collections.Generic`

F.3.1 Mutable lists

`List`, `LinkedList`

F.3.2 Stacks

`Stack`

F.3.3 Queues

`Queue`

F.3.4 Sets and dictionaries

`HashSet`, and `Dictionary` from

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