Learning to program with F#

Jon Sporring

September 20, 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 20, 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
· statements
· state

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.

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

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].

· Structured programming
· Object-orientered programming
· objects

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

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, gettingStartedStump.fsx:
A simple demonstration script.**

```
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 $\boxed{\text{box}}$:

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`.

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

· format string
· string

For the `printfn` function, the format string may be any string, but if it contains format character sequences, such as `%A`, then the values following the format string are substituted. The format string must match the value *type*, that is, here `c` is of type integer, whereas the format string `%A` matches many types. · 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: 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: 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*. · 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]

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: 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: 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: 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: 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: 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

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.

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

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

· 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: 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.

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

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

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

Listing 5.10: 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: 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: 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: 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: 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: 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: 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: 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: 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: 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: 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: 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: 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: 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 of any of the integer types, but in this case, F# casts an *exception*,

· exception

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

Table 5.5: Boolean exclusive or truth table.

Listing 5.25: 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: 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: 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: 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: 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: 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) -`

Advice

357.9) < 1e-10.

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

5.7 Programming intermezzo

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

$$v = \sum_{i=0}^{n} b_i 2^i \qquad (5.1)$$

For example $10011_2 = 1 + 2 + 16 = 19$. From decimal to binary is a little more complex, but a simple divide-by-two algorithm exists. The key to understanding the divide-by-two algorithm is to realize that when you divide a number by two, then that is equivalent to shifting its binary representation 1 to the right. E.g., $10 = 1010_2$ and $10/2 = 5 = 101_2$. Odd numbers have $b_0 = 1$, e.g., $11_{10} = 1011_2$ and $11_{10}/2 = 5.5 = 101.1_2$. Hence, if we divide any number by two and get a non-integer number, then its least significant bit was 1. Another way to express this is that the least significant bit is the remainder after integer division by two. Sequential application of this idea leads directly to the divide-by-two algorithm. E.g., if we were to convert the number 11_{10} on decimal form to binary form we would perform the following steps:

$$\begin{tikzcd}
11 \text{ div } 2 = 5, \arrow[d] \arrow[dr] & 11 \text{ rem } 2 = 1 \\
5 \text{ div } 2 = 2, \arrow[d] \arrow[dr] & 5 \text{ rem } 2 = 1 \\
2 \text{ div } 2 = 1, \arrow[d] \arrow[dr] & 2 \text{ rem } 2 = 0 \\
1 \text{ div } 2 = 0, & 1 \text{ rem } 2 = 1
\end{tikzcd}$$

Here we used div and rem to signify the integer division and remainder operators. The algorithms stops, when the result of integer division is zero. Reading off the remainder from below and up we find the sequence 1011_2, which is the binary form of the decimal number 11_{10}. Using interactive mode, we can calculate the same as,

Listing 5.35: Converting the number 11_{10} to binary form.

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

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

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. As an example, consider the problem,

Problem 6.1:

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

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

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

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

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

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

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

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

```
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has discriminant 4.0 and solutions -1.0 and 1.0
```

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

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

· lightweight syntax

The use of identifiers is central in programming. For F# not to be confused by built-in functionality, identifiers must follow a specific grammar: An identifier must start with a letter, but can be followed by zero or more of letters, digits, and a range of special characters except SP, LF, and CR (space, line feed, and carriage return). An identifier must not be a keyword or a reserved-keyword listed in Figures 6.1. An identifier is a name for a constant, an expression, or a type, and it is defined by the following EBNF:

Keywords:

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.

Reserved keywords for possible future use:

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.

Symbolic keywords:

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

Reserved symbolic keywords for possible future:

~ and `.

Figure 6.1: List of (possibly future) keywords and symbolic keywords in F#.

Listing 6.2: Identifiers

```
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.

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

Expressions has an enormous variety in how they may be written, we will in this book gradually work through some of the more important facets. For this we will extend the EBNF notation with ellipses: `...`, to denote that what is shown is part of the complete EBNF production rule. E.g., the part of expressions, we will discuss in this chapter is specified in EBNF by,

Listing 6.3: Simple expressions.

```
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 (*assignment*)

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

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

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

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

In the following sections, we will work through this bit by bit.

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: Value binding expression.

```
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" (3.0 ** p)
```

```
9.0
```

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: 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 when F# determines the value bound to a name, it looks left and upward in the program text for the "let" statement defining it, 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
```

F# also has to option of using dynamic scope, where the value of a binding is defined by when it is used, and this will be discussed in Section 6.5.

Scopes are given levels, and scopes may be nested, where the nested scope has a level one lower than its parent.[1] F# distinguishes between the top and lower levels, and at the top level in the lightweight syntax, redefining values is not allowed, 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 parentheses, we create a *block*, i.e., a *nested scope*, and then redefining is allowed, e.g., · block · nested scope

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

Listing 6.14, 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. Bindings inside are not available outside the nested scope, e.g.,

Listing 6.15, letValueScopeNestedScope.fsx:
Bindings inside a scope are not available outside.

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

```



/Users/sporring/repositories/fsharpNotes/src/letValueScopeNestedScope.fsx
    (6,19): error FS0039: The value or constructor 'q' is not defined
```

Nesting is a natural part of structuring code, e.g., through function definitions to be discussed in Section 6.2 and flow control structures to be discussed in Chapter 8. Blocking code by nesting is a key concept for making robust code that is easy to use by others without the user necessarily needing to know the details of the inner workings of a block of code.

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

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 F# interprets the above as `let p = 3 in let p = 4 in (printfn "%A" p; printfn "%A" p)`. Had we intended to print the two different values of `p`, the we should have create a block as,

**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
```

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

6.2 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: Function binding expression

```
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: 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 specification is sufficient, and we could just have specified the type of the result,

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

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

or even just one of the arguments,

Listing 6.21, letFunctionAlterantive2.fsx:
Just one type is often enough for F# to infer the rest.

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

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

· 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: 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 therefore can be of any type, and which F# therefore calls `'a`, and the type of the second element, `y`, can also be of any type and not necessarily the same as `x`, so it is called `'b`. Finally the result is the same type as `y`, whatever it is. This is an example of a *generic function*, since it will work on any type. · generic function

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

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

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

let a = 1.0
let b = 0.0
let c = -1.0
let xp = solution a b c +1.0
let xn = solution a b c -1.0
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, but requires a final expression, which will be returned to the caller of the function. Note also that since the function `discriminant` is defined in the nested scope of `solution`, then `discriminant` cannot be called outside `solution`, since the scope ends before `let a = 1.0`.

Lexical scope and function definitions can be a cause of confusion as the following example shows,[2] · 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
```

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

Here, the value-binding for `a` is redefined, after it has been used to define a helper function `f`. So which value of `a` is used, when we later apply `f` to an argument? To resolve the confusion, remember that value-binding is lexically defined, i.e., the binding `let f z = a * 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`. Advice

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

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, functionDeclarationCurrying.fsx:
A function can be defined as a subset of another by Currying.

```
let mul x y = x*y
let timesTwo = mul 2.0
printfn "%g" (mul 5.0 3.0)
printfn "%g" (timesTwo 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, `timesTwo` is a function of 1 argument being the second argument of `mul`. This notation is called *currying* in tribute of Haskell Curry, and Currying is often used in functional programming, but generally **currying should be used carefully, since currying may seriously reduce readability of code.** · currying Advice

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

Listing 6.29, procedure.fsx:
A procedure is a function that has no return value, and in F# returns "()".

```
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. **Procedural thinking is useful for encapsulation, but is prone to side-effects and should be minimized by being replaced by functional thinking.** · encapsulation · side-effects Advice

6.3 User-defined operators

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

**Listing 6.30, addOperatorNFunction.fsx:
Operators have function equivalents.**

```
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, but in F# names of user-defined operators are limited by the following simplified EBNF:

Listing 6.31: 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 6.32, operatorDefinitions.fsx:
Operators may be (re)defined by their function equivalent.

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

```
13
16
true
2
```

Operators beginning with `*` must use a space in its definition, `( *` in order for it not to be confused with the beginning of a comment `(*`, see Chapter 7 for more on comments in code.

Beware, redefining existing operators lexically redefines all future uses of the operators for all types, hence **it is not a good idea to redefine operators, but better to define new.** In Chapter 20 we will discuss how to define type specific operators including prefix operators. Advice

6.4 The Printf function

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

Listing 6.33: `printf` statement.

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

where a `formatString` is a string (simple or verbatim) with placeholders. The function `printf` prints `formatString` to the console, where all `placeholder` has been replaced by the value of the corresponding argument formatted as specified, e.g., in `printfn "1 2 %d" 3` the `formatString` is `"1 2 %d"`, and the placeholder is `%d`, and the `printf` replaced the placeholder with the value of the corresponding argument, and the result is printed to the console, in this case `1 2 3`. Possible formats for the placeholder are,

Listing 6.34: Placeholders in **`formatString`** for `printf` functions.

```
placeholder = "%%" | ("%" [flags] [width] ["." precision] specifier) (* No
    spaces between rules *)
flags = ["0"] ["+"] [SP] (* No spaces between rules *)
width = ["-"] ("*" | [dInt]) (* No spaces between rules *)
specifier = "b" | "d" | "i" | "u" | "x" | "X" | "o" | "e" | "E" | "f" | "F" |
    "g" | "G" | "M" | "O" | "A" | "a" | "t"
```

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

Table 6.1: Printf placeholder string

There are specifiers for all the basic types and more as elaborated in Table 6.1. The placeholder can be given a specified with, either by setting a specific integer, or using the `*` character, indicating that the with is given as an argument prior to the replacement value. Default is for the value to be right justified in the field, but left justification can be specified by the `-` character. For number types, you can specify their format by: `"0"` for padding the number with zeros to the left, when right justifying the number; `"+"` to explicitly show a plus sign for positive numbers; `SP` to enforce a space, where there otherwise would be a plus sign for positive numbers. For floating point numbers, the precision integer specifies the number of digits displayed of the fractional part. Examples of some of these combinations are,

Listing 6.35, 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#, examples of their use are,

Listing 6.36, printfExampleAdvance.fsx:
Custom format functions may be used to specialize output.

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

```
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 to predefine a `formatString` as, e.g., `let str = "hello %s" in printf str "world"` will be a type error.

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.

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 with the following syntax,[3]

· "`<-`"

Listing 6.37: Value reassignment for mutable variables.

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

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 6.38, 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
```

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

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

· "`<-`"

Listing 6.39: It is a common error to mistake "=" and "<-" lexemes for mutable variables.

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

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

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 is 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 6.40, 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, for example,

Listing 6.41, 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
```

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

Listing 6.42: Returning a mutable variable returns its value.

```
> let g () =
-   let mutable y = 0
-   y
- printfn "%d" (g ());;
0

val g : unit -> int
val it : unit = ()
```

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

Variables implement dynamic scope, e.g., in comparison with the lexical scope, where the value of an identifier depends on *where* it is defined, dynamic scope depends on, *when* it is used. E.g., the script in Listing 6.25 defines a function using lexical scope and returns the number `6.0`, however, if `a` is made `mutable`, then the behaviour is different:

Listing 6.43, 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.

It is possible to work with mutable variables but through a special technique called *encapsulation*. · encapsulation
E.g., in the following example the we create a counter as an encapsulated mutable variable,

Listing 6.44, mutableAssignIncrementEncapsulation.fsx:
Local mutable content can be indirectly accessed outside its scope.

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

```
1
2
3
```

This works because the line `let mutable counter = 0` is only executed once, when the function `incr` is defined. This is also an example of a side-effect. **Encapsulation of mutable data is good programming practice, but avoiding mutable data all together is better practice.** Advice

F# has a variation of mutable variables called *reference cells*. Reference cells have built-in function `ref` and operators "`!`" and "`:=`", · reference cells

Listing 6.45, refCell.fsx:
Reference cells are variants of mutable variables.

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

```
0
1
```

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*, where variable changes are performed across independent scopes. · side-effects The `incr` example in Listing 6.44 is an example of a side-effect. Another example is,

Listing 6.46, mutableAssignReturnSideEffect.fsx:
Intertwining independent scopes is typically a bad idea.

```
let updateFactor factor =
  factor := 2

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

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

```
6
```

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

Listing 6.47, mutableAssignReturnWithoutSideEffect.fsx:
A solution of Listing 6.46 avoiding side-effects.

```
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 have their use, but should in general be avoided at almost all costs, and in general it is advised to **minimize the use of side effects**. Advice

Reference cells gives rise to an effect called *aliasing*, where two or more identifiers refer to the same data as illustrated by the following example: · aliasing

Listing 6.48, refCellAliasing.fsx:
Aliasing can cause surprising results and should be avoided.

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

```
1, 1
2, 2
```

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

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 that lead to it. Documentation is most often a mixture between in-code documentation and accompanying documents. Here, we will focus on in-code documentation, 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,

Listing 7.1: Comments.

```
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 7.2, commentExample.fsx:
Code with XML comments.

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

/// <summary>Find x when 0 = ax^2+bx+c.</summary>
/// <remarks>Negative discriminant are not checked.</remarks>
/// <example>
///   The following code:
///   <code>
///     let a = 1.0
///     let b = 0.0
///     let c = -1.0
///     let xp = (solution a b c +1.0)
///     printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
///   </code>
///   prints <c>0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7</c> to the console.
/// </example>
/// <param name="a">Quadratic coefficient.</param>
/// <param name="b">Linear coefficient.</param>
/// <param name="c">Constant coefficient.</param>
/// <param name="sgn">+1 or -1 determines the solution.</param>
/// <returns>The solution to x.</returns>
let solution a b c sgn =
  let d = discriminant a b c
  (-b + sgn * sqrt d) / (2.0 * a)

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

```
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 7.3, 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 7.4, An XML file generated by `fsharpc`.

```
<?xml version="1.0" encoding="utf-8"?>
<doc>
<assembly><name>commentExample</name></assembly>
<members>
<member name="M:CommentExample.solution(System.Double,System.Double,System
    .Double,System.Double)">
 <summary>Find x when 0 = ax^2+bx+c.</summary>
 <remarks>Negative discriminant are not checked.</remarks>
 <example>
   The following code:
   <code>
     let a = 1.0
     let b = 0.0
     let c = -1.0
     let xp = (solution a b c +1.0)
     printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
   </code>
   prints <c>0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7</c> to the console.
 </example>
 <param name="a">Quadratic coefficient.</param>
 <param name="b">Linear coefficient.</param>
 <param name="c">Constant coefficient.</param>
 <param name="sgn">+1 or -1 determines the solution.</param>
 <returns>The solution to x.</returns>
</member>
<member name="M:CommentExample.discriminant(System.Double,System.Double,
    System.Double)">
<summary>
 The discriminant of a quadratic equation with parameters a, b, and c
</summary>
</member>
</members>
</doc>
```

The extracted XML is written in C# type by convention, since F# is part of the Mono and .Net framework that may be used by any of the languages using Assemblies. Besides the XML inserted in the script, the XML has added `<?xml ...>` header, `<doc>`, `<assembly>`, `<members>`, and `<member >` tags. The header and the `<doc>` tag are standards for XML. The extracted XML is geared towards documenting big libraries of codes and thus highlights the structured programming 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`, whose primary use is to perform a syntax analysis of an assembly and generate a scaffold XML structure for an accompanying document. With the `-i` flag, it is further possible to include the in-code comments as initial descriptions in the XML. The XML may be updated gracefully by `mdoc` as the code develops, without destroying manually entered documentation in the accompanying documentation. Finally, the XML may be exported to HTML

The primary use of the `mdoc` command is to 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,

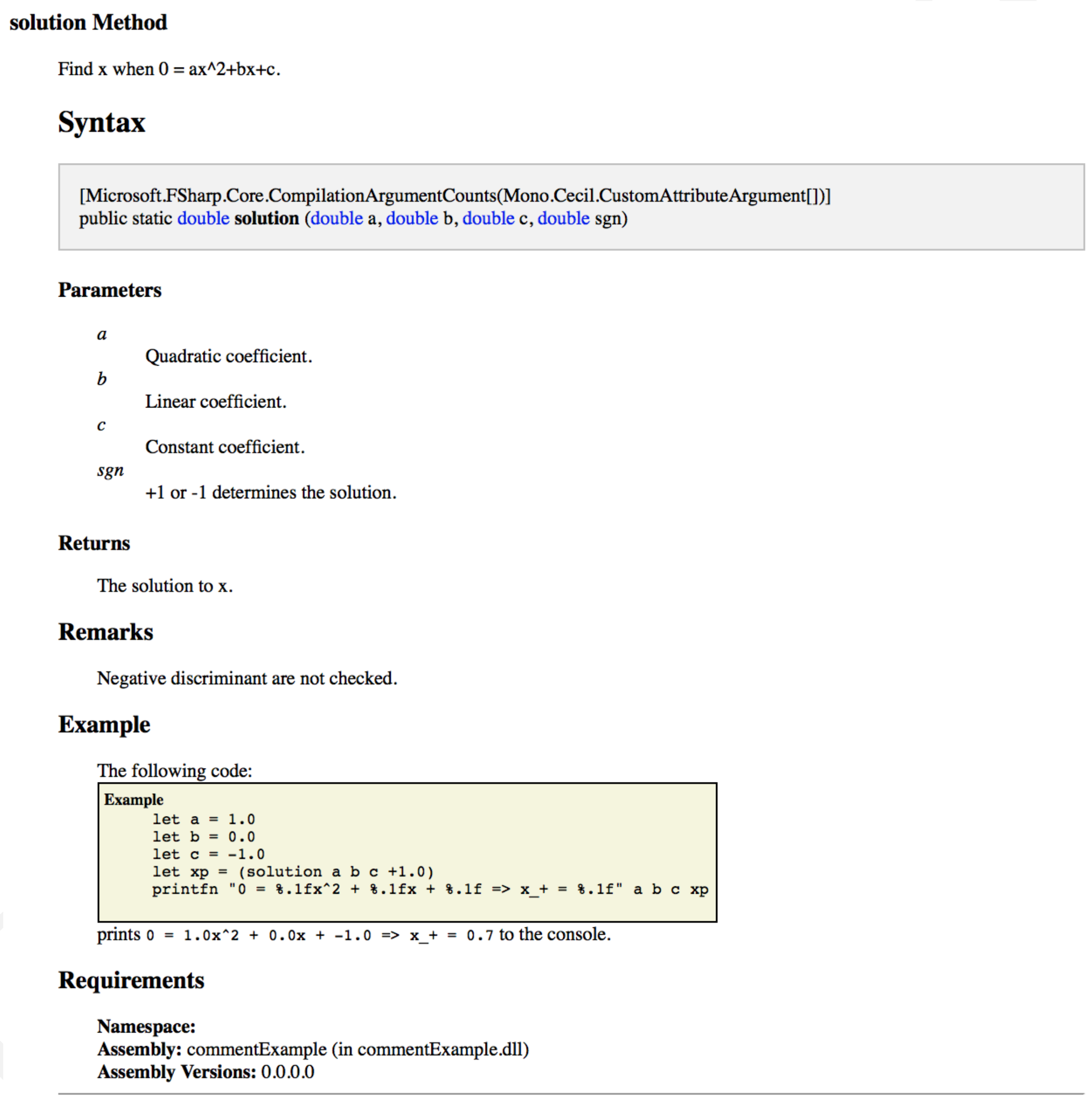
· Hyper Text Markup Language
· HTML

Listing 7.5, 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 determines the solution.

Returns

The solution to x.

Remarks

Negative discriminant are not checked.

Example

The following code:

Example
```
let a = 1.0
let b = 0.0
let c = -1.0
let xp = (solution a b c +1.0)
printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
```
prints `0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7` to the console.

Requirements

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

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

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: Expressions for controlling the flow of execution.

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

8.1 For and while loops

Many programming constructs need to be repeated, and F# contains many structures for repetition such as the "`for`" and "`while`" loops, which have the syntax,

Listing 8.2: `for`- and `while`-loops.

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

As an example, consider counting from 1 to 10 with a "*`for`*"-loop, · "`for`"

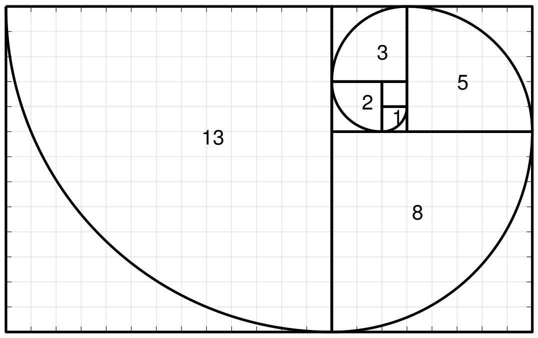


Figure 8.1: The Fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive Fibonacci numbers, and the curve in each square is the circular arc with radius of the square it is drawn in

Listing 8.3: Counting from 1 to 10 using a "for"-loop.

```
> for i = 1 to 10 do printf "%d " i done;
- 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 "`()`" like the `printf` functions. Using lightweight syntax the block following the "*do*" keyword up to and including the "*done*" keyword 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, see Listing 8.3.

```
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 calculates the n'th Fibonacci number.

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

Listing 8.5, fibFor.fsx:
The n'th Fibonacci number is the sum of the previous 2.

```
let fib n =
  let mutable prev = 1
  let mutable current = 1
  let mutable next = 0
  for i = 3 to n do
    next <- current + prev
    prev <- current
    current <- next
  next

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). For the fib(4) we only need fib(3) and fib(2), hence we may disregard fib(1). Thus we realize, that we can cyclicly update the previous, current and next values by shifting values until we have reached the desired fib(n).

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 8.3 example, it would look somewhat like,

· "while"

Listing 8.6, countWhile.fsx:
Count to 10 with a counter variable.

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

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

or equivalently using the lightweight syntax,

Listing 8.7, countWhileLightweight.fsx:
Count to 10 with a counter variable using lightweight syntax, see Listing 8.6.

```
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 prev = 1
  let mutable current = 1
  let mutable next = 0
  while next <= n do
    next <- prev + current
    prev <- current
    current <- next
  prev

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) " i a
printf "\n"
```

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

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 8.2:
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 the value of n, and one option is to use conditional expressions. Conditional expression has the syntax, The grammar for conditional expressions is,

Listing 8.10: Conditional expressions.

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

and an example using conditional expressions to solve the above problem is,

Listing 8.11, conditionalLightweight.fsx:
Using conditional expression to generate different strings.

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

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

```
"I owe 3 apples"
"I owe 1 apple"
"I have no apples"
"I have 1 apple"
"I have 2 apples"
"I have 10 apples"
```

The 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 identical type, such that regardless which branch is chosen, then the type of the result of the conditional expression is the same. The result of the conditional expression is the first branch, for which its condition was true.

· "if"
· "elif"
· conditions
· "then"
· "else"
· branches

The sentence structure and its variants gives rise to a more compact solution, since the language to be returned to the user is a variant of "I have/or no/number apple(s)", i.e., under certain conditions should the sentence use "have" and "owe" etc.. So we could instead make decisions on each of these sentence parts and then built the final sentence from its parts. This is accomplished in the following example:

Listing 8.12, conditionalLightweightAlt.fsx:
Using sentence parts to construct the final sentence.

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

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

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

```
"I owe 3 apples"
"I owe 1 apple"
"I have no apples"
"I have 1 apple"
"I have 2 apples"
"I have 10 apples"
```

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

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

8.3 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, for this there is a technical solution, and we will just concern ourselves with the logics of using recursion for programming. The syntax for defining recursive functions in F# is,

· recursive function

Listing 8.13: Recursive functions.

```
expr = ...
  | "let" "rec" functionDefn {"and" functionDefn} "in" expr
```

An example of a recursive function that counts from 1 to 10 similarly to Listing 8.3 is,[1]

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 at the end of the recursion branch, then this is a *tail-recursive* function. Most compilers achieve high efficiency in terms of speed and memory, so **prefer tail-recursion whenever possible.** Using recursion to calculate the Fibonacci number as Listing 8.5.

· stopping criterium
· tail-recursive
Advice

Listing 8.15, 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
```

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

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.** Advice

Functions that recursively call each other are called *mutually recursive* functions. F# offers the "let"-"rec"-"and" notation for co-defining mutually recursive functions. As an example, consider the function `even : int -> bool`, which returns true if its argument is even and false otherwise, and the opposite function `odd : int -> bool`. A mutually recursive implementation of these functions can be developed from the following statements: `even 0 = true`, `odd 0 = false`, and `even n = odd (n-1)`: · mutually recursive

Listing 8.16, mutuallyRecursive.fsx:
Using mutual recursion to implement even and odd functions.

```
let rec even x =
  if x = 0 then true
  else odd (x - 1)
and odd x =
  if x = 0 then false
  else even (x - 1);;

let w = 5;
printfn "%*s %*s %*s" w "i" w "even" w "odd"
for i = 1 to w do
  printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
    i  even   odd
    1 false  true
    2  true false
    3 false  true
    4  true false
    5 false  true
```

Notice that in the lightweight notation used here, that the "and" must be on the same indentation level as the original "let".

Without the "and" keyword, F# will return an error at the definition of `even`. However, it is possible to implement mutual recursion by using functions as an argument, e.g.,

Listing 8.17, mutuallyRecursiveAlt.fsx:
Mutual recursion without the “and” keyword needs a helper function.

```
let rec evenHelper (notEven: int -> bool) x =
  if x = 0 then true
  else notEven (x - 1)

let rec odd x =
  if x = 0 then false
  else evenHelper odd (x - 1);;

let even x = evenHelper odd x

let w = 5;
printfn "%*s %*s %*s" w "i" w "Even" w "Odd"
for i = 1 to w do
  printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
    i  Even   Odd
    1 false  true
    2  true false
    3 false  true
    4  true false
    5 false  true
```

But, Listing 8.16 is clearly to be preferred over Listing 8.17.

In the above we used the `even` and `odd` function problems to demonstrate mutual recursion. There is, of course, a much simpler solution, which does not use recursion at all:

Listing 8.18, parity.fsx:
A better way to test for parity without recursion.

```
let even x = (x % 2 = 0)
let odd x = not (even x)
```

which is to be preferred anytime as the solution to the problem.

8.4 Programming intermezzo

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

Problem 8.3:

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

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

Listing 8.19, dec2bin.fsx:
Using integer division and remainder to write any positive integer on binary form.

```
let dec2bin n =
  let rec dec2binHelper n =
    let mutable v = n
    let mutable str = ""
    while v > 0 do
      str <- (string (v % 2)) + str
      v <- v / 2
    str

  if n < 0 then
    "Illegal value"
  elif n = 0 then
    "0b0"
  else
    "0b" + (dec2binHelper n)

printfn "%4d -> %s" -1 (dec2bin -1)
printfn "%4d -> %s" 0 (dec2bin 0)
for i = 0 to 3 do
  printfn "%4d -> %s" (pown 10 i) (dec2bin (pown 10 i))
```

```
  -1 -> Illegal value
   0 -> 0b0
   1 -> 0b1
  10 -> 0b1010
 100 -> 0b1100100
1000 -> 0b1111101000
```

Another solution is to use recursion instead of the “`while`” loop:

Listing 8.20, dec2binRec.fsx:
Using recursion to write any positive integer on binary form, see also Listing 8.19.

```
let dec2bin n =
  let rec dec2binHelper n =
    if n = 0 then ""
    else (dec2binHelper (n / 2)) + string (n % 2)

  if n < 0 then
    "Illegal value"
  else
    "0b" +
    if n = 0 then
      "0"
    else
      dec2binHelper n

printfn "%4d -> %s" -1 (dec2bin -1)
printfn "%4d -> %s" 0 (dec2bin 0)
for i = 0 to 3 do
  printfn "%4d -> %s" (pown 10 i) (dec2bin (pown 10 i))
```

```
  -1 -> Illegal value
   0 -> 0b0
   1 -> 0b1
  10 -> 0b1010
 100 -> 0b1100100
1000 -> 0b1111101000
```

Listing 8.19 is a typical imperative solution, where the states `v` and `str` are iteratively updated until `str` finally contains the desired solution. Listing 8.20 is a typical functional programming solution, to be discussed in Part III, where the states are handled implicitly as new scopes created by recursively calling the helper function. Both solutions have been created using a local helper function, since both solutions require special treatment of the cases $n < 0$ and $n = 0$.

Let us compare the two solutions more closely: The computation performed is the same in both solutions, i.e., integer division and remainder is used repeatedly, but since the recursive solution is slightly shorter, then one could argue that it is better, since shorter programs typically have fewer errors. However, shorter program also typically means that understanding them is more complicated, since shorter programs often rely on realisations that the author had while programming, which may not be properly communicated by the code nor comments. Speedwise, there is little difference between the two methods: 10,000 conversions of `System.Int32.MaxValue`, i.e., the number 2,147,483,647, takes about 1.1 sec for both methods on an 2,9 GHz Intel Core i5 machine.

Notice also, that in Listing 8.20, the prefix `"0b"` is only written once. This is advantageous for later debugging and updating, e.g., if we later decide to alter the program to return a string without the prefix or with a different prefix, then we would only need to change one line instead of two. However, the program has gotten slightly more difficult to read, since the string concatenation operator and the `if` statement are now intertwined. There is thus no clear argument for preferring one over the other by this argument.

Proving that Listing 8.20 computes the correct sequence is easily done using the induction proof technique: The result of `dec2binHelper 0` is clearly an empty string. For calls to `dec2binHelper n` with $n > 0$, we check that the right-most bit is correctly converted by the remainder function, and that this string is correctly concatenated with `dec2binHelper` applied to the remaining bits. A simpler way to state this is to assume that `dec2binHelper` has correctly programed, so that in the body of `dec2binHelper`, then recursive calls to `dec2binHelper` returns the correct value. Then we only need to check that the remaining computations are correct. Proving that Listing 8.19 calculates the correct sequence essentially involves the same steps: If $v = 0$ then the "`while`" loop is skipped, and the result is the initial value of `str`. For each iteration of the "`while`" loop, assuming that `str` contains the correct conversions of the bits up till now, we check that the remainder operator correctly concatenates the next bit, and that `v` is correctly updated with the remaining bits. We finally check that the loop terminates, when no more 1-bits are left in `v`. Comparing the two proofs, the technique of assuming that the problem has been solved, i.e., that recursive calls will work, helps us focus on the key issues for the proof. Hence, we conclude that the recursive solution is most elegantly proved, and thus preferred.

Part V

Appendix

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

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

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 there are infinitely many numbers. Reals are widely used for calculation, but since any computer only has finite memory, it is impossible to represent all possible reals. Hence, any computation performed on a computer with reals must rely on approximations. *IEEE 754 double precision floating-point format* (*binary64*), known as a *double*, is a standard for representing an approximation of reals using 64 bits. These bits are divided into 3 parts: sign, exponent and fraction,

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

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

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

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

Table B.1: ASCII

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

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

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			¬	¼	Ì	Ü	ì	ü
0d			SHY	½	Í	Ý	í	ý
0e			®	¾	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.

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#. 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
· terminal symbols
· production rules

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

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

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

```
number = digit { 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";
signedNumber = ( "+" | "−" ) digit, { digit };
```

here `signedNumber` is a number with a mandatory sign, such as `+5` and `−3`.

'`" ... "`' 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:

Rule	Description
`s | t` ↔ `t | s`	| is commutative
`r | (s | t)` ↔ `(r | s) | t` ↔ `r | s | t`	| is associative
`(r, s) t` ↔ `r (s, t)` ↔ `r, s, t`	concatenation is associative

`r, (s | t)` ↔ `r, t | r, s`
`(r | s), t` ↔ `r, t | r, t`
	concatenation is distributive over |

`[s | t]` ↔ `[t] | [s]`	
`[[s]]` ↔ `[s]`	`[]` is idempotent
`{{s}}` ↔ `{s}`	`{}` is idempotent

where `r`, `s`, and `t` are production rules or terminals. Precedence for the EBNF symbols are,

Symbol	Description
`[]`, ...	Bracket and quotation mark pairs
−	except
`,`	concatenate
|	option
=	define
`;`	terminator

in order of precedence, such that bracket and quotation mark pairs has higher precedence than −.

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 (*assignment*)
  | 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"  functionDefn "in" expr (*binding a function or operator*)
  | "let" "rec" functionDefn {"and" functionDefn} "in" expr (*recursive functions
   *)
  | "fun" argumentPats "->" expr (*anonymous function*)
  | "if" expr "then" expr {"elif" expr "then" expr} ["else" expr] (*conditional*)
  | "while" expr "do" expr ["done"] (*while loop*)
  | "for" ident "=" expr "to" expr "do" expr ["done"] (*simple for loop*)
  | "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

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

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

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

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**

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