Learning to program with F#

Jon Sporring

August 2, 2016

Contents

Chapter 2

Introduction

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

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

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

Implement the plan: Implementation is the act of transforming a program design into a code. A crucial part of any implementation is choosing which programming language to use. Also, the solution to many problems will have a number of implementations which vary in how much code they require, to which degree they rely on external libraries, which programming style 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 the steps in Pólya's list are almost always to be performed, but the order of the steps and the number of times each step is performed varies. [1]
This book focusses on 3 fundamentally different approaches to programming:

· Imperative programming
· statements
· state

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

· Declarative programming

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

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

· Functional programming
· functions
· expressions

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

· Structured programming
· Object-orientered programming
· objects

Most programs follows a single programming paradigm as, e.g., one of the above, but are a mix. Nevertheless, this book will treat each paradigm separately to emphasize their advantages and disadvantages.
This book uses F# also known as Fsharp, which is a functional first programming language that also supports imperativ and object oriented programming. It was originally developed for Microsoft's .Net platform, but is available as open source for many operating systems through Mono. As an introduction to programming, F# is a young programming language still under development, with syntax that at times is a bit complex, but it offers a number of advantages:

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

Indentation for scope F# uses indentation to indicate scope.

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

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

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

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

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

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

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

Part I

F# basics

Chapter 3

Executing F# code

3.1 Source code

F# is a functional first programming language that also supports imperativ 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 `fshparc` 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*

`.fsi` A *signature file*

`.fsx` A *script file*

`.fsscript` Same as `.fsx`

`.exe` An *executable file*

- 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*.

- script-fragments
- modules

3.2 Executing programs

Programs may either be executed by the interpreter or by compiling and executing the compiled code. In `Mono` the interpreter is called `fsharpi` and can be used in 2 ways: interactively, where a user enters 1 or more script-fragments separated by the `";;"` lexeme, or to execute a script file treated as a single script-fragment. To illustrate the difference, consider the following program, which declares a value `a` to be the decimal value 3.0 and finally print it to the console:

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

An interactive session is obtained by starting the console, typing the `fsharpi` command, typing the lines of the program, and ending the script-fragment with the `";;"` lexeme:

```
$ fsharpi

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

For help type #help;;

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

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

> #quit;;
```

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

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

Finally, compiling and executing the code is performed as,

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

Both the interpreter and the compiler translates the source code into a format, which can be executed by the computer. While the compiler performs this translation once and stores the result in the executable file, the interpreter translates the code every time the code is executed. Thus, to run the program again with the interpreter, then it must be retranslated as "`$fsharpi gettingStartedStump.fsx`", but since the program has been compiled, then the compile-execute only needs the be re-executed "`$ mono gettingStartedStump.exe`". On a Macbook Pro, with a 2.9 Ghz Intel Core i5, the time the various stages takes for this script are.

Command	Time
`fsharpi gettingStartedStump.fsx`	1.88s
`fsharpc gettingStartedStump.fsx`	1.90s
`mono randomTextOrder0.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* a method for debugging programs.

· type inference
· debugging
· unit-testing

Chapter 4

Quick-start guide

Programming is the art of solving problems by writing a program to be executed by a computer. For example, to solve the following problem,

> What is the sum of 357 and 864?

we have written the following program in F#,

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

```
1221
```

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

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

To solve the program we made program consisting of several lines, where each line was a *statement*. The first statement `let a = 357` used the *`let` keyword* to *bind* the value 357 to the name `a`. Likewise, we bound the value 864 to the name `b`, but to the name `c` we bound the result of evaluating the *expression* `a + b`. That is, first the value `a + b` was calculated by substituting the names of `a` and `b` with their values to give the expression, `357 + 864`, then this expression was evaluated by adding the values to give, `1221`, and this value was finally bound to the name `c`. The last line printed the value of `c` to the console followed by a newline (LF possibly preceded by CR, see Appendix B.1) with the `printfn` function. Here `printfn` is a function of 2 arguments: `"%A"` and `c`. Notice, that in contrast to many other languages, F# does not use parentheses to frame the list of arguments, nor does it use commas to separate them. In general, the `printfn` function always has 1 or more arguments, and the first is a *format string*. A *string* is a sequence of characters starting and ending with double quotation marks. E.g., `let s = "this is a string of characters"` binds the string `"this is..."` to the name `s`. For the `printfn` function, the format string may be any string, but if it contains format character sequences, such as `%A`, then the values following the format string are substituted. The format string must match the value *type*, that is, here `c` is of type integer, whereas the format string `%A` matches any type.

- statement
- `let`
- keyword
- binding
- expression
- format string
- string
- type

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

- type declaration
- type inference

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

```
val a : int = 357

> let b = 864;;

val b : int = 864

> let c = a + b;;

val c : int = 1221

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

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

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

· `val` Advice

Were we to solve a slightly different problem,

> What is the sum of 357.6 and 863.4?

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

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

```
1221.0
```

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

On the surface, this could appear as an almost negligible change, but the set of integers and the set of real numbers (floats) require quite different representations, in order to be effective on a computer, and as a consequence, the implementation of their operations such as addition are very different. Thus, although the response is an integer, it has type `float`, which is indicated by `1221.0` which is not the same as `1221`. F# is very picky about types, and generally does not allow types to be mixed. E.g., in an interactive session,

```
> let a = 357;;

val a : int = 357

> let b = 863.4;;

val b : float = 863.4

> let c = a + b;;

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

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

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

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

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

· lexical scope

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

```


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

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

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

```
> let a = 357;;

val a : int = 357

> let a = 864;;

val a : int = 864
```

Listing 4.6: fsharpi, Names may be reused when separated by the lexeme `;;`.

then apparently rebinding is valid. The difference is that the *`;;` lexeme* defines a new nested *scope*.[2] A lexeme is a letter or a word, which the F# considers as an atomic unit. Scopes can be *nested*, and in F# a binding may reuse names in a nested scope, in which case the previous value is *overshadowed*. I.e., attempting the same without `;;` between the two let statements results in an error, e.g.,

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

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

  let a = 864;;
  ----^

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

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

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

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

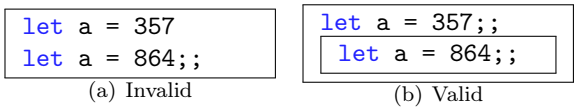
· function

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

```
1221
```

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

[1] **When command is omitted, then error messages have unwanted blank lines.**
[2] **Language change: Spec 4.0 p. 15.1 talks about blocks instead of scopes.**



(a) Invalid (b) Valid

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

Entering the function into an interactive session will illustrate the inferred type, the function `sum` has: `val sum : x:int * y:int -> int`, by which is meant that `sum` is a mapping from the set product of integers with integers into integers. Type inference in F# may cause problems, since the type of a function is inferred in the context, in which it is defined. E.g., in an interactive session, defining the sum in one scope on a single line will default the types to integers, F#'s favorite type, which will give an error, if it in a nested scope is to be used for floats,

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

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

> let c = sum 357.6 863.4;;

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

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

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

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

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

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

Listing 4.10: fsharpi, Defining a function together with its use, makes F# infer the appropriate types.

In this chapter, we have scratched the surface of learning how to program by concentrating on a number of key programming concepts and how they are expressed in the F# language. In the following chapters, we will expand the description of F# with features used in all programming approaches.

Chapter 5

Using F# as a calculator

5.1 Literals and basic types

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

- type
- literal

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

Listing 5.1: fsharpi, Typing the number 3.

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

- `int`
- `it`

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

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

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

- `float`
- `char`
- `string`
- basic types

Humans like to use the *decimal number* system for representing numbers. Decimal numbers are *base* 10 means that for a number consisting of a sequence of digits separated by a *decimal point*, where each *digit* can have values $d \in \{0, 1, 2, \ldots, 9\}$, and the value, which each digit represents is proportional to its position. The part befor the decimal point is called the *whole part* and the part after is called the *fractional part* of the number. The whole part without a decimal point and a fractional part is called an *integer number*. As an example `35.7` is a decimal number, whose value is $3 \cdot 10^1 + 5 \cdot 10^0 + 7 \cdot 10^{-1}$. In F# a decimal number is called a *floating point number* and in this text we use *Extended Backus-Naur Form* (*EBNF*) to describe the grammar of F#, the decimal number just described is given as,

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

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

Metatype	Type name	Description
Boolean	bool	Boolean values true or false
Integer	**int**	Integer values from -2,147,483,648 to 2,147,483,647
	byte	Integer values from 0 to 255
	sbyte	Integer values from -128 to 127
	int8	Synonymous with byte
	uint8	Synonymous with sbyte
	int16	Integer values from -32768 to 32767
	uint16	Integer values from 0 to 65535
	int32	Synonymous with int
	uint32	Integer values from 0 to 4,294,967,295
	int64	Integer values from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
	uint64	Integer values from 0 to 18,446,744,073,709,551,615
	nativeint	A native pointer as a signed integer
	unativeint	A native pointer as an unsigned integer
Real	**float**	64-bit IEEE 754 floating point value from $-\infty$ to ∞
	double	Synonymous with float
	single	A 32-bit floating point type
	float32	Synonymous with single
	decimal	A floating point data type that has at least 28 significant digits
Character	**char**	Unicode character
	string	Unicode sequence of characters
None	**unit**	No value denoted
Object	**obj**	An object
Exception	**exn**	An exception

Table 5.1: List of basic types. The most commonly used types are highlighted in bold. For at description of integer see Appendix A.1, for floating point numbers see Appendix A.2, for ASCII and Unicode characters see Appendix B, for objects see Chapter 18, and for exceptions see Chapter 10.

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

meaning that a `dDigit` is either "0" or "1" or ... or "9", an `dInt` is 1 or more `dDigit`, and a `dFloat` is 1 or more digits, a dot and 0 or more digits. There is no space between the digits and between digits and the dot. So `3`, `049` are examples of integers, `34.89` `3.` are examples of floats, while `.5` is neither.

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

· scientific notation

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

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

The basic unit of information in almost all computers is the binary digit or *bit* for short. A *binary number* consists of a sequence of binary digits separated by a decimal point, where each digit can have values $b \in \{0, 1\}$, and the base is 2. E.g., the binary number $101.01_2 = 1{\cdot}2^2{+}0{\cdot}2^1{+}1{\cdot}2^0{+}0{\cdot}2^{-1}{+}1{\cdot}2^1 = 5.25$. Binary numbers are closely related to *octal* and *hexadecimal numbers*, where octals uses 8 as basis and can be written in binary using 3 bits, while hexadecimal numbers uses 16 as basis and can be written in binary using 4 bits. Octals and 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

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

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

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

· character
· Unicode
· code point

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

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

A *string* is a sequence of characters enclosed in double quotation marks,[2]

· string

[1] **Spec-4.0 p.28:** `char-char` **is missing option** `unicodegraph-long`

[2] **Spec-4.0 p. 28-29:** `simple-string-char` **is undefined,** `string-elem` **is unused.**

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

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

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

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

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

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

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

Listing 5.3: fsharpi, Examples of string literals.

The response is shown in double quotation marks, which are not part of the string.
F# supports *literal types*, where the type of a literal is indicated as a prefix og suffix as shown in the Table 5.3. Examples are, · literal type

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

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

Table 5.3: List of literal type. No spacing is allowed between the literal and the prefix or suffix. There must not be a space between the number and the literal name. The [] notation is for lists, see Chapter 9. *bignum is not a basic type and does not yet have an implementation for dInt ("Q"\|"R"\|"Z"\|"N"\|"G") in Mono.

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

Listing 5.4: fsharpi, Named and implied literals.

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

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

Listing 5.5: fsharpi, Examples of a string literal.

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

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

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

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

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

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

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

Listing 5.7: fsharpi, Casting booleans.

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

· member
· class
· namespace

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

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

Listing 5.8: fsharpi, Fractional part is removed by downcasting.

Here we type casted to a lesser type, in the sense that integers is a subset of floating point numbers, and this is called *downcasting*. The opposite is called *upcasting* and is often non-destructive, as Listing 5.6 showed, where an integer was casted to a float while retaining its value. As a side note, *rounding* a number $y.x$, where y is the *whole part* and x is the *fractional part*, is the operation of mapping numbers in the interval $y.x \in [y.0, y.5)$ to y and $y.x \in [y.5, y+1)$ to $y+1$. This can be performed by downcasting as follows,

· downcasting
· upcasting
· rounding
· whole part
· fractional part

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

Listing 5.9: fsharpi, Fractional part is removed by downcasting.

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

5.2 Operators on basic types

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

· expression
· infix operator

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

Listing 5.10: expressionArithmetic

[3] **Spec-4.0 Section 4.3: const is missing uint8, int8 nativeint, unativeint.**

Operator	op1	op2	Expression	Result	Description

`op1 + op2`	ints
	floats
	chars
	strings
		ints
		floats
		chars
		strings
			`5 + 2`
			`5.0 + 2.0`
			`'a' + 'b'`
			`"ab" + "cd"`
				`7`
				`7.0`
				`'\195'`
				`"abcd"`
					Addition
					
					Addition of codes
					Concatenation

`op1 - op2`	ints floats	ints floats	`5 - 2` `5.0 - 2.0`	`3` `3.0`	Subtraction
`op1 * op2`	ints floats	ints floats	`5 * 2` `5.0 * 2.0`	`10` `10.0`	Multiplication
`op1 / op2`	ints floats	ints floats	`5 / 2` `5.0 / 2.0`	`2` `2.5`	Integer division Division
`op1 % op2`	ints floats	ints floats	`5 % 2` `5.0 % 2.0`	`1` `1.0`	Remainder
`op1 ** op2`	floats	floats	`5.0 ** 2.0`	`25.0`	Exponentiation
`op1 && op2`	bool	bool	`true && false`	`false`	boolean and
`op1 || op2`	bool	bool	`true || false`	`false`	boolean or
`op1 &&& op2`	ints	ints	`0b1010 &&& 0b1100`	`0b1000`	bitwise bool and
`op1 ||| op2`	ints	ints	`0b1010 ||| 0b1100`	`0b1110`	bitwise boolean or
`op1 ^^^ op2`	ints	ints	`0b1010 ^^^ 0b1101`	`0b0111`	bitwise boolean exclusive or
`op1 <<< op2`	ints	ints	`0b00001100uy <<< 2`	`0b00110000uy`	bitwise shift left
`op1 >>> op2`	ints	ints	`0b00001100uy >>> 2`	`0b00000011uy`	bitwise and
`+op1`	ints floats		`+3` `+3.0`	`3` `3.0`	identity
`-op1`	ints floats		`-3` `-3.0`	`-3` `-3.0`	negation
`not op1`	bool		`not true`	`false`	boolean negation
`~~~op1`	ints		`~~~0b00001100uy`	`0b11110011uy`	bitwise boolean negation

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

Recursion means that a rule or a function is used by the rule or function itself in its definition. See Part III for more on recursion. Infix notation means that the *operator* `op` appears between the two *operands*, and since there are 2 operands, it is a *binary operator*. As the grammar shows, the operands themselves can be expressions. Examples are `3+4` and `4+5+6`. Some operators only takes one operand, e.g., `-3`, where `-` here is used to negate a 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 shown in Table 5.4 and 5.5 and a range of mathematical functions shown in Table 5.6. Arithmetic on various types will be discussed in detail in the following sections. [4]

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

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

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

Listing 5.11: fsharpi, A simple arithmetic expression.

[4] **minor comment on indexing and slice-ranges.**

Operator	op1	op2	Expression	Result	Description
`op1 < op2`	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`	
`op1 > op2`	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`	
`op1 = op2`	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`	
`op1 <= op2`	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`	
`op1 >= op2`	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`	
`op1 <> op2`	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 5.5: Comparison operators on basic types. Ints, floats, chars, and strings means all built-in integer types etc..

Type	Function name	Example	Result	Description
Ints and floats	`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 5.6: Predefined functions for arithmetic operations

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

Table 5.7: Some common operators, their precedence, and their associativity. Rows are ordered from highest to lowest precedences, such that `op * op` has higher precedence than `op + op`. Operators in the same row has same precedence. Full table is given in Table D.1.

Here, the addition and multiplication functions are shown in *infix notation* with the *operator* lexemes `+` and `*`. To arrive at the resulting value 23, F# has to decide in which order to perform the calculation. There are 2 possible orders, `3 + (4 * 5)` or `(3 + 4)* 5`, which gives different results. For integer arithmetic, the correct order is of course to multiply before addition, and we say that multiplication takes *precedence* over addition. Every atomic operation that F# can perform is ordered in terms of its precedences, and for some common built-in operators shown in Table 5.7, the precedence is shown by the order they are given in the table. Associativity implies the order in which calculations are performed for operators of same precedence. For some operators and type combinations association matters little, e.g., multiplication associates to the left and exponentiation associates to the right, e.g., in[5]

· infix notation
· operator
· precedence
· boolean or
· boolean and

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

Listing 5.12: fsharpi, Precedences rules define implicite parentheses.

[5] **Spec-4.0, Table 18.2.1 appears to be missing boolean 'and' and 'or' operations. Section 4.4 seems to be missing &&& and ||| bitwise operators.**

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

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

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

5.3 Boolean arithmetic

Boolean arithmetic is the basis of almost all computers and particularly important for controlling program flow, which will be discussed in Chapter 8. Boolean values are one of 2 possible values, true or false, which is also sometimes written as 1 and 0. Two basic operations on boolean values are '*and*' often also written as multiplication, and '*or*' often written as addition, and '*not*' often written as a bar above the value. All possible combination of input on these values can be written on tabular form, known as a *truth table*, shown in Table 5.8. That is, the multiplication and addition are good mnemonics for remembering the result of the 'and' and 'or' operators. In F# the values `true` and `false` are used, and the operators `&&` for 'and', `||` for 'or', and the function `not` for 'not', such that the above table is reproduced by,

- · and
- · or
- · not
- · truth table

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

val it : unit = ()
```

Listing 5.13: fsharpi, Boolean operators and truth tables.

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

5.4 Integer arithmetic

The set of integers is infinitely large, but since all computers have limited resources, it is not possible to represent it in their entirety. The various integer types listed in Table 5.1 are finite subset reduced by limiting their ranges. Although `bignum` is theoretically unlimited, the biggest number representable is still limited by computer memory. An in-depth description of integer implementation can be found in Appendix A. The type `int` is the most common type.
Table 5.4, 5.5, and 5.6 gives examples operators and functions pre-defined for integer types. Notice that fewer functions are available for integers than for floating point numbers. For most addition, subtraction, multiplication, and negation the result straight forward. However, performing arithmetic operations on integers requires extra care, since the result since they may cause *overflow*, *underflow*, e.g., the range of the integer type `sbyte` is $[-128 \ldots 127]$, which causes problems in the following example,

· overflow
· underflow

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

Listing 5.14: fsharpi, Adding integers may cause overflow.

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

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

Listing 5.15: fsharpi, Subtracting integers may cause underflow.

I.e., we were expecting a negative number, but got a postive number instead.
The overflow error in Listing 5.14 can be understood in terms of the binary representation of integers: In binary, $130 = 10000010_2$, and this binary pattern is interpreted differently as `byte` and `sbyte`,

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

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

That is, for signed bytes, the left-most bit is used to represent the sign, and since the addition of $100 = 01100100_2$ and $30 = 00011110_b$ 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 *integer division*, which discards the fractional part after division, and the *remainder* operator calculates the remainder after integer division, e.g.,

· integer division
· remainder

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

Listing 5.17: fsharpi, Integer division and remainder operators.

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

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

a	b	a xor b
0	0	0
0	1	1
1	0	1
0	1	0

Table 5.9: Boolean exclusive or truth table.

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

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

And we see that integer division of 7 by 3 followed by multiplication by 3 is less 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

```
> 3/0;;
System.DivideByZeroException: Attempted to divide by zero.
  at <StartupCode$FSI_0002>.$FSI_0002.main@ () <0x68079f8 + 0x0000e> in <
      filename unknown>:0
  at (wrapper managed-to-native) System.Reflection.MonoMethod:InternalInvoke (
      System.Reflection.MonoMethod,object,object[],System.Exception&)
  at System.Reflection.MonoMethod.Invoke (System.Object obj, BindingFlags
      invokeAttr, System.Reflection.Binder binder, System.Object[] parameters,
       System.Globalization.CultureInfo culture) <0x1a7c270 + 0x000a1> in <
      filename unknown>:0
Stopped due to error
```

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

The output looks daunting at first sight, but the first and last line of the error message are the most important parts, which tells us what exception was cast and why the program stopped. The middle are technical details concerning which part of the program caused this, and can be ignored for the time being. Exceptions are a type of *run-time error*, and are treated in Chapter 10

· run-time error

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

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

Listing 5.20: fsharpi, Integer exponent function.

which is equal to 2^5.
For binary arithmetic on integers, the following operators are available: `op1 <<< op2`, which shifts the bit pattern of `op1` `op2` positions to the left insert 0's to right; `op1 >>> op2`, which shifts the bit pattern of `op1` `op2` positions to the right insert 0's to left; `op1 &&& op2`, Bitwise 'and', returns the result of taking the boolean 'and' operator position-wise; `op ||| op`, Bitwise 'or', as 'and' but using the boolean 'or' operator; and `op1 ~~~ op1`, Bitwise xor, which is returns the result of the boolean 'xor' operator defined by the truth table in Table 5.9.

· 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. The various floating point types listed in Table 5.1 are finite subset

reduced by sampling the space of reals. An in-depth description of floating point implementations can be found in Appendix A. The type `float` is the most common type.
Table 5.4, 5.5, and 5.6 gives examples operators and functions pre-defined for floating point types. For most addition, subtraction, multiplication, divisions, and negation the result straight forward.
The remainder operator for floats calculates the remainder after division and discarding the fractional part,

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

Listing 5.21: fsharpi, Floating point division and remainder operators.

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

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

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

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

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

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

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

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

Listing 5.24: fsharpi, Floating point arithmetic has finite precision.

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

5.6 Char and string arithmetic

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

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

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

I.e., the code point difference between upper and lower case for any alphabetical character 'a' to 'z' is constant, hence we can change case by adding or subtracting the difference between any corresponding character. Unfortunately, this does not generalize to characters from other languages.
A large collection of operators and functions exist for `string`. The most simple is concatenation using, e.g.,

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

Listing 5.26: fsharpi, Example of string concatenation.

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

· .[]

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

Listing 5.27: fsharpi, String indexing using square brackets.

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

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

Listing 5.28: fsharpi, String length attribute and string indexing.

Notice, since index counting starts at 0, and the string length is 7, then the index of the last character is 6. An alternative notation for indexing is to use the property `Char`, and in the example `''abcdefg''.[3]` is the same as `a.Char 3`. The is a long list of built-in functions in `System.String` for working with strings, some of which will be discussed in Chapter E.1.
The *dot notation* is an example of Structured programming, where technically speaking, the string `"abcdefg"` is an immutable *object* of *class* `string`, and `[]` is an object *method* and `Length` is a property. For more on object, classes, and methods see Chapter 18.

· dot notation
· object
· class
· method

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

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

Chapter 6

Constants, functions, and variables

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

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

we use the quadratic formula from elementary algebra,

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

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

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

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

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

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

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

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

· lightweight syntax

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

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

Figure 6.1: List of keywords in F#.

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

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

line feed, and carriage return). An identifier must not be a keyword or a reserved-keyword listed in Figures 6.1, 6.2, 6.3, and 6.4
For characters in the Basic Latin Block, i.e., the first 128 code points alias ASCII characters, an ident is,

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

Thus, examples of identifiers are `a`, `theCharacter9`, `Next_Word`, `_tok`. The for the full definition, `letter = Lu | Ll | Lt | Lm | Lo | Nl` and `specialChar = Pc | Mn | Mc | Cf`, which referes to the Unicode general categories described in Appendix B.3, and there are currently 19.345 possible Unicode code points in the `letter` category and 2.245 possible Unicode code points in the `specialChar` category. Binding expressions to identifiers is done with the keyword `let`, using the following simplified syntax:

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

which will be discussed in the following.[1]

[1] **Spec-4.0 Section 6.6, function-or-value-defns, possible Mono deviation from specification: let rec functionDefn and functionDefn requires newline before and.**

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

Figure 6.3: List of symbolic keywords in F#.

~ and `.

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

6.1 Values

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

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

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

· let
· :
· mutable
· lexically
· in

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

```
8.0
```

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

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

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

```
9.0
```

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

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

· lightweight syntax

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

```
9.0
```

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

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

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

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

Listing 6.5: fsharpi, Interactive mode also responds inferred types.

[2] https://coders-corner.net/2013/11/12/lexical-scope-vs-dynamic-scope/

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

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

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

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

Here, the left-hand-side is defined to be an identifier of type float, while the right-hand-side is a literal of type integer.
An expression can be a sequence of expressions separated by the lexeme `;`, e.g.,

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

```
2.0
9.0
```

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

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

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

```
2.0
9.0
```

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

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

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

```
 4
```

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

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

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

[3] **Drawings would be good to describe scope**

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

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

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

· block
· nested scope

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

```
4
```

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

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

· overshadows

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

```
4
```

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

In both cases we used indentation, which is good practice, but not required here. Lowering level is a natural part of function definitions to be discussed in Section 6.2 and flow control structures to be discussed in Chapter 8.
Defining blocks is useful for controlling the extend of a lexical scope of bindings. For example, adding a second `printfn` statement,

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

```
4
4
```

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

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

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

```
 4
 3
```

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

[4] **spacing in lstinline mode after double quotation mark is weird.**

Here the lexical scope of `let p = 4 in ...` is for the nested scope, which ends at `)`, returning to the lexical scope of `let p = 3 in ....` Alternatively, the `begin` and `end` keywords could equally have been used.
[5]

6.2 Non-recursive functions

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

· encapsulate code

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

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

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

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

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

and we see that the function is interpreted to have the type `val sum : x:float -> y:float -> float`. The `->` lexeme means a mapping between sets, in this case floats. The function is also a higher order function, to be discussed in detail below, and here it suffices to think of `sum` as a function that takes 2 floats as argument and returns a float.
Not all types need to be declared, just sufficient for F# to be able to infer the types for the full statement. In the example, one sufficient specification is, and we could just have specified the type of the result,

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

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

or even just one of the arguments,

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

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

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

· operator
· operand

[5] **Remember to say something about interactive scripts and the ;; lexeme and scope**

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

```
1221.0
```

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

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

```
> let second x y = y
- let a = second 3 5
- printfn "%A" a
- let b = second "horse" 5.0
- printfn "%A" b;;
5
5.0

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

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

Here the function `second` does not use the first argument, `x` which is any type called `'a`, and the type of the second element, `y`, is also any type an not necessarily the same as `x`, so it is called `'b`. Finally the result is the same type as `y`, whatever it is. This is an example of a *generic function*. · generic function
A function may contain a sequence of expressions, but must return a value. E.g., the quadratic formula may written as,

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

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

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

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

Here we used the lightweight syntax, where the `=` identifies the start of a nested scope, and F# identifies the scope by indentation. The amount of space used for indentation is does not matter, but all lines following the first must use the same. The scope ends before the first line with the previous indentation or none. Notice how the last expression is not bound to an identifier, but is the result of the function, i.e., in contrast to many other languages, F# does not have an explicit keyword for returning values. Note also that since the function `determinant` is defined in the nested scope of `solution`, then `determinant` cannot be called outside `solution`, since the scope ends before `let a = 1.0`.
Lexical scope and function definitions can be a cause of confusion as the following example shows,[6] · lexical scope

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

```
6.0
```

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

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

Advice

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

· anonymous function

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

```
5
```

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

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

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

```
18
```

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

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

Advice

Functions may be declared from other functions

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

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

```
15
6
```

Listing 6.24: functionDeclarationTupleCurrying.fsx -

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

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

```
15
6
```

Listing 6.25: functionDeclarationCurrying.fsx -

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

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

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

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

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

In F# this is automatically given the unit type as return value. Procedural thinking is useful for *encapsulation* of scripts, but is prone to *side-effects* and should be minimized by being replaced by functional thinking. More on side-effects in Section 6.5. which also does not have a return value. Procedural thinking is useful for encapsulation, but is prone to side-effects and should be minimized by being replaced by functional thinking. · encapsulation · side-effects
[8]

6.3 User-defined operators

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

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

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

Listing 6.27: addOperatorNFunction.fsx -

[8] **Remember examples of return of functions, no arguments `()`, and wildcard patterns as arguments `_`.**

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

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

Listing 6.28: Grammar for infix and prefix lexemes

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

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

```
13
16
true
2
```

Listing 6.29: operatorDefinitions.fsx -

Beware, redefining existing operators lexically redefines all future uses of operator for all types, hence *it is not a good idea to redefine operators, but better to define new.*[9] In Chapter /refchap:oop we will discuss how to define type specific operators including prefix operators. Advice

Operators beginning with `*` must use a space in its definition, `( *` in order for it not to be confused with the beginning of a comment `(*`. [10]

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

[10] **this requires comments to be describe previously!**

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

Table 6.1: Printf placeholder string

6.4 The Printf function

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

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

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

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

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

```
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   "
```

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

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

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

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

Listing 6.31: printfExampleAdvance.fsx -

The `%A` is special in that all built-in types including tuples, lists, and arrays to be discussed in Chapter 9 can be printed using this formatting string, but notice that the formatting performed includes the named literal string. The two formatting strings `%t` and `%a` are options for user-customizing the formatting, and will not be discussed further.
Beware, `formatString` is not a `string` but a `Printf.TextWriterFormat`, so `let str = "hello %s"in printf str "world"` will be a type error.
The family of `printf` is shown in Table 6.2. The function `fprintf` prints to a stream, e.g., `stderr` and `stdout`, of type `System.IO.TextWriter`. Streams will be discussed in further detail in Chapter 12. The function `failwithf` is used with exceptions, see Chapter 10 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)` [13] `· ignore`

[11] **Mono seems to have a bug, `printfn "%.1g" 3.13;;` and `printfn "%.1f" 3.13;;` produces different number of digits.**
[12] **Spec-4.0 %s and %b are missing in Section 3.1.16.**
[13] **Mono: `bprintf` and `kprintf` are undefined.**

Function	Example	Description
`printf` `printfn`	`printf "%d apples"3`	Prints to the console, i.e., `stdout` as `printf` and adds a newline.

`fprintf`
 	
`fprintfn`
	`fprintf stream "%d apples"3`	Prints to a stream, e.g., `stderr` and `stdout`, which would be the same as `printf` and `eprintf`.
		as `fprintf` but with added newline.

`eprintf` `eprintfn`	`eprintf "%d apples"3`	Print to `stderr` 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.

6.5 Variables

The `mutable` in `let` bindings means that the identifier may be rebound to a new value using the lexeme, e.g.,[14] · <-

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

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

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

```
5
-3
```

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

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

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

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

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

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

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

[14] **Discussion on heap and stack should be added here.**

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

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

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

I.e., once the type of an identifier has been declared or inferred, then it cannot be changed.
A typical variable is a counter of type integer, and a typical use of counters is to increment them, i.e., erasing a new value to be one more that its previous value. For example,

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

```
5
6
```

Listing 6.35: mutableAssignIncrement.fsx - Variable increment is a common use of variables.

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

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

```
1
2
3
```

Listing 6.36: mutableAssignIncrementEncapsulation.fsx -

[15] Here the output of `incr` is an anonymous function, that takes no argument, increments the variable of `incr` and returns the new value of the counter. This construction is called *encapsulation*, since the variable `counter` is hidden by the function `incr` from the user, i.e., the user need not be concerned with how the increment operator is implemented and the variable name used by `incr` does not clutter the scope where it is used. · encapsulation
Variables implement dynamic scope, e.g., in comparison with the lexical scope, where the value of an identifier depends on which line in the program, an identifier is defined, dynamic scope depends on, when it is used. E.g., the script in Listing 6.21 defines a function using lexical scope and returns the number `6.0`, however, if `a` is made `mutable`, then the behaviour is different:

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

[15] **Explain why this works!**

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

```
8.0
```

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

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

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

```
0
```

Listing 6.38: mutableAssignReturnValue.fsx -

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

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

```


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

Listing 6.39: mutableAssignReturnVariable.fsx -

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

· reference cells

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

```
0
3
```

Listing 6.40: mutableAssignReturnRefCell.fsx -

That is, the `ref` function creates a reference variable, the '`!`' and the '`:=`' operators reads and writes its value. Reference cells are in some language called pointers, and their use is strongly discouraged, since they may cause *side-effects*, which is the effect that one function changes the state of another, such as the following example demonstrates,[16]

· side-effects

[16] **Discuss side-effects!**

```
let updateFactor factor =
  factor := 2

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

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

```
6
```

Listing 6.41: mutableAssignReturnSideEffect.fsx -

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

```
let updateFactor () =
  2

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

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

```
6
```

Listing 6.42: mutableAssignReturnWithoutSideEffect.fsx -

Here there can be no doubt in `multiplyWithFactor` that the value of 'a' is changing. Side-effects do have their use, but should in general be avoided at almost all costs, and in general it is advised to refrain from using ref cells.
[17]

[17] **Add something about mutable functions**

Chapter 7

In-code documentation

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

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

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

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

That is, text framed as a `blockComment` is still parsed by F# as keywords and basic types implying that `(* a comment (* in a comment *)*)` and `(* "*)" *)` are valid comments, while `(* " *)` is invalid.[1]
The F# compiler has an option for generating *Extensible Markup Language* (*XML*) files from scripts using the C# documentation comments tags[2]. The XML documentation starts with a triple-slash `///`, i.e., a lineComment and a slash, which serves as comments for the code construct, that follows immediately after. XML consists of tags which always appears in pairs, e.g., the tag "tag" would look like `<tag> ... </tag>`. The F# accept any tags, but recommends those listed in Table 7.1. If no tags are used, then it is automatically assumed to be a `<summary>`. An example of a documented script is,

- Extensible Markup Language
- XML

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

/// <summary>Find x when 0 = ax^2+bx+c.</summary>
/// <remarks>Negative determinants are not checked.</remarks>
/// <example>
///   The following code:
///   <code>
```

[1] **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.

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

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

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

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

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

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

This results in an XML file with the following content,

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

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

An accompanying program in the Mono suite is `mdoc`, which primary use is to perform a syntax analysis of an assembly and generate a scaffold XML structure for an accompanying document. With the `-i` flag, it is further possible to include the in-code comments as initial descriptions in the XML. The XML may be updated gracefully by `mdoc` as the code develops, without destroying manually entered documentation in the accompanying documentation. Finally, the XML may be exported to HTML

The primary use of the `mdoc` command is to analyze compiled code and generate an empty XML structure with placeholders to describe functions, values, and variables. This structure can then be updated and edited as the program develops. The edited XML files can then be exported to *Hyper Text Markup Language* (*HTML*) files, which can be viewed in any browser. Using the console, all of

· Hyper Text Markup Language
· HTML

solution Method

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

Syntax

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

Parameters

a
Quadratic coefficient.
b
Linear coefficient.
c
Constant coefficient.
sgn
+1 or -1 indicating which solution is to be calculated.

Returns

The solution to x.

Remarks

Negative determinants are not checked.

Example

The following code:

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

results in `0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7` printed to the console.

Requirements

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

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

this is accomplished by,

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

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

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,

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

8.1 For and while loops

Many programming constructs need to be repeated. The most basic example is counting, e.g., from 1 to 10 with a *for*-loop,[1] · for

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

val it : unit = ()
```

Listing 8.1: fsharpi, Counting from 1 to 10 using a `for`-loop.

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

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

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

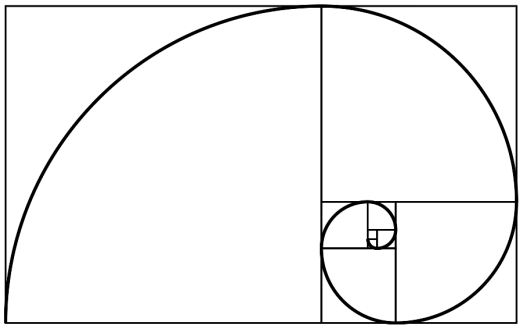


Figure 8.1: The Fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive Fibonacci numbers, and the curve in each square is the circular arc with radius of the square it is drawn in. Figure by Dicklyon `https://commons.wikimedia.org/w/index.php?curid=3730979`

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

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

· do
· done

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

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

Listing 8.2: countLightweight.fsx - Counting from 1 to 10 using a `for`-loop.

A more complicated example is,

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

The Fibonacci numbers is the series of numbers $1, 1, 2, 3, 5, 8, 13 \ldots$, 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,

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

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

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

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

Listing 8.3: fibFor.fsx - The n'th Fibonacci number as the sum of the previous 2 numbers, which are sequentially updated from 3 to n.

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

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

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

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

primeFactorCheck 10
```

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

Listing 8.4: primeCheck.fsx - Checking whether a given number has remainder zero after division by some low prime numbers.

In this example, the variable `i` runs through the elements of a list, which will be discussed in further detail in Chapter 9.
The *while*-loop is simpler than the `for`-loop and does not contain a builtin counter structure. Hence, if we are to repeat the count-to-10 program from Listing 8.1 example, it would look somewhat like, · `while`

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

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

Listing 8.5: countWhile.fsx - Count to 10 with a counter variable.

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

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

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

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

Listing 8.6: fibWhile.fsx - Search for the largest Fibonacci number less than a specified number.

Thus, `while`-loops are most often used, when the number of iteration cannot easily be decided, when entering the loop.
Both `for`- and `while`-loops are often associated with variables, i.e., values that change while looping. If one mistakenly used values and rebinding, then the result would in most cases be of little use, e.g.,

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

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

Listing 8.7: forScopeError.fsx - Lexical scope error. While rebinding is valid F# syntax, has little effect due to lexical scope.

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

8.2 Conditional expressions

Consider the task,

> Write a function that given n writes the sentence, "I have n apple(s)", where the plural 's' is added appropriately.

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

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

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

Listing 8.8: Using conditional expression to generate different strings.

The grammar for conditional expressions is,

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

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

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

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

Listing 8.9: Lightweight syntax allows for making blocks of code by indentation in order to make code more for easy to read.

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

8.2.1 Programming intermezzo

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

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

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

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

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

Listing 8.10: dec2bin.fsx - Using integer division and remainder to convert any positive integer to binary form.

8.3 Pattern matching

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

· pattern matching

> Write a function that given n writes the sentence, "I have n apple(s)", where the plural 's' is added appropriately.

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

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

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

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

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

Listing 8.11: matchWith.fsx - Using the `match`-keywordwith programming construct to vary calculation based on the input value.

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

· `match`
· `with`
· `when`

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

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

Using lightweight syntax, the rules may be put on separate lines but must start in the column, where the `match` starts or greater.[2] Match with can only take one identifier, but this can be tuples for matching with combinations of identifiers, see Chapter 9 for more on tuples. A `match` expression is general but is most often seen as the initial part of a function definition. This is so common, that F# has a special syntax integrating function definitions and match with expressions using the *function* keyword,

· `function`

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

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

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

[2] **Spec-4.0 weirdness: Offside rule for match is different for function.**

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

Listing 8.12: functionKeyword.fsx - Function definition and `match` expressions are integrated using the `function` keyword. Compare with Listing 8.11

Comparing with Listing 8.11 notice that the function definition does not explicitly name an argument but assumes one, following the `function` follows immediately the rules, and the wildcard pattern `_` is replaced with an identifier without any guards, which thus matches everything. Replacing the wildcard pattern with a name has the advantage that this name can be used locally in the expression belonging to this rule, i.e., it acts as a `let n =` on the implicit argument of the function. Implicit arguments makes the code hard to read and, thus *the use of function definitions with the keyword `function` should be avoided.* Advice

8.4 Recursive functions

Recursion is a central concept in F#. A *recursive function* is a function, which calls itself. From a compiler point of view, this is challenging, since the function is used before the compiler has completed its analysis. However, this there is a technical solution for, and we will just concern ourselves with the logics of using recursion for programming. An example of a recursive function that counts from 1 to 10 similarly to Listing 8.1 is,[3] · recursive function

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

prt 1 10
```

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

Listing 8.13: countRecursive.fsx - Counting to 10 using recursion.

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

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

Using recursion to calculate the Fibonacci number as Listing 8.3.

```
let rec fib n =
  match n with
    i when i < 1 -> 0
    | i when i = 1 -> 1
    | i -> fib (n - 1) + fib (n - 2)

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

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

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

Listing 8.14: fibRecursive.fsx - The n'th Fibonacci number using recursive.

Here we used the fact that including $\text{fib}(0) = 0$ in the Fibonacci series also produces it using the rule $\text{fib}(n) = \text{fib}(n-2) + \text{fib}(n-1),\ n \geq 0$, which allowed us to define a function that is well defined for the complete set of integers. I.e., a negative argument returns 0. This is a general advice: *make functions that fails gracefully.* The recursive definition allows for recursive value definitions and defining several values and functions in one expression. Recursive values is particularly useful for defining infinite sequences, see Section 9.4.

Advice

Chapter 9

Ordered series of data

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

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

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

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

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

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

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

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

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

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

9.1 Tuples

Tuples are unions of immutable types, · tuple

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

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

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

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

Listing 9.2: fsharpi, Definition of a tuple.

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

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

val deconstructNPrint : 'a * 'b * 'c -> unit
```

[2]**Spec-4.0: grammar for list and array expressions are subsets of computation list and array expressions.**

```
val it : unit = ()
```

Listing 9.3: fsharpi, Definition of a tuple.

In this a function is defined that takes 1 argument, a 3-tuple, and which is bound to a tuple with 3 named members. Since we used the `%A` placeholder in the `printfn` function, then the function is generic and can be called with 3-tuples of different types. Note, *don't confuse a function of n arguments with a function of an n-tuple.* The later has only 1 argument, and the difference is the `,`'s. Another example is `let solution a b c = ...`, which is the beginning of the function definition in Listing 9.1. It is a function of 3 arguments, while `let solution (a, b, c)= ...` would be a function of 1 argument, which is a 3-tuple. Functions of several arguments makes currying easy, i.e., we could define a new function which fixes the quadratic term to be 0 as `let solutionToLinear = solution 0.0`, that is, without needing to specify anything else. With tuples, we would need the slightly more complicated, `let solutionToLinear (b, c)= solution (0.0, b, c)`.

Advice

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

· obfuscation

Advice

9.2 Lists

Lists are unions of immutable values of the same type and have a more flexible structure than tuples. Its grammar follows *computation expressions*, which is very rich and shared with arrays and sequences, and we will delay a discussion on most computation expressions to Section 9.4, and here just consider a subset of the grammar:

· list

· computation expressions

```
listOrArrayList =  expr | expr ";" listOrArrayList
range-exp = expr ".." expr ["." expr]
expr = ...
  | "[" (listOrArrayList | ... | range-expr) "]" (* computation list expression *)
  | ...
```

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

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

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

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

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

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

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

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

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

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

Listing 9.8: listSlicing.fsx - Examples of list slicing. Compare with Listing 5.27.

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

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

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

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

```
1 2 3 4 5
```

Listing 9.9: listPatternMatching.fsx - Examples of list concatenation, indexing.

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

· pattern matching

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

A list of course names and grades is `[("name1", grade1); ("name2", grade2); ...]`. Let's take a recursive solution. First problem will be to iterate through the list. For this we can use pattern matching similarly to Listing 9.9 with `(name, grade)::rest`. The second problem will be to calculate the average. The average grade is the sum all grades and divide by the number of grades. Assume that we already have made a function, which calculates the `sum` and `n`, the sum and number of elements, for `rest`, then all we need is to add `grade` to the `sum` and `1` to `n`. For an empty list, `sum` and `n` should be `0`. Thus we arrive at the following solution,

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

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

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

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

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

Listing 9.10: avgGradesRec.fsx - Calculating a list of average grades using recursion and pattern matching.

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

```
let rec square a =
```

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

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

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

Listing 9.11: listSquare.fsx - Using pattern matching and list appending elements to lists.

This is a prototypical functional programming style solution, and which uses the `::` for 2 different purposes: First the list `[1 .. 10]` is first matched with `1 :: [2 .. 10]`, and then we assume that we have solved the problem for `square rest`, such that all we need to do is append `1*1` to the beginning output from `square rest`. Hence we get, `square [1 .. 10]` ↷ `1 * 1 :: square [2 .. 10]` ↷ `1 * 1 :: (2 * 2 :: square [3 .. 10])` ↷ ... `1 * 1 :: (2 * 2 :: ... 10 * 10 :: [])`, where the stopping criterium is reached, when the `elm :: rest` does not match with a, hence it is empty, which does match the wildcard pattern `_`. More on functional programming in Section 16
The basic properties and members of lists are summarized in Table 9.1. In addition, lists have many other built-in functions, such as functions for converting lists to arrays and sequences,

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

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

Listing 9.12: listConversion.fsx - The `List` module contains functions for conversion to arrays and sequences.

These and more will be discussed in Chapter E and Part III.
It is possible to make multidimensional lists as lists of lists, e.g.,

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

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

Listing 9.13: listMultidimensional.fsx - A ragged multidimensional list, built as lists of lists, and its indexing.

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

· ragged multidimensional list

9.3 Arrays

One dimensional arrays or just arrays for short are mutable lists of the same type and follow a similar syntax as lists. Its grammar follows *computation expressions*, which will be discussed in Section 9.4. Here we consider a subset of the grammar:

· computation expressions

```
listOrArrayList =  expr | expr ";" listOrArrayList
range-exp = expr ".." expr ["." expr]
expr = ...
```

Function name	Example	Description

`Length`	`> [1; 2; 3].Length;;`
	`val it : int = 3`
	`> let a = [1; 2; 3] in a.Length;;`
	`val it : int = 3`
		The number of elements in a list

`List.Empty`	`> let a : int list = List.Empty;;`
	
	`val a : int list = []`
	
	`> let b = List<int>.Empty;;`
	
	`val b : int list = []`
		An empty list of specified type

`IsEmpty`	`> [1; 2; 3].IsEmpty;;`
	`val it : bool = false`
	`> let a = [1; 2; 3] in a.IsEmpty;;`
	`val it : bool = false`
		Compare with the empty list

`Item`	`> [1; 2; 3].Item 1;;`
	`val it : int = 2`
	`> let a = [1; 2; 3] in a.Item 1;;`
	`val it : int = 2`
		Indexing

`Head`	`> [1; 2; 3].Head;;`
	`val it : int = 1`
	`> let a = [1; 2; 3] in a.Head;;`
	`val it : int = 1`
		The first element in the list. Exception if empty.

`Tail`	`> [1; 2; 3].Tail;;`
	`val it : int list = [2; 3]`
	`> let a = [1; 2; 3] in a.Tail;;`
	`val it : int list = [2; 3]`
		The list except its first element. Exception if empty.

`Cons`	`> list.Cons (1, [2; 3]);;`
	`val it : int list = [1; 2; 3]`
	`> 1 :: [2; 3];;`
	`val it : int list = [1; 2; 3]`
		Append an element to the front of the list

`@`	`> [1] @ [2; 3];;`
	`val it : int list = [1; 2; 3]`
	`> [1; 2] @ [3; 4];;`
	`val it : int list = [1; 2; 3; 4]`
	`> [1; 2] @ [3];;`
	`val it : int list = [1; 2; 3]`
		Concatenate two lists

Table 9.1: Basic properties and members of lists. The syntax used in `List<int>.Empty` ensures that the empty list is of type `int`.

```
  | "[|" (listOrArrayList | ... | range-expr) "|]" (* computation array expression
      *)
  | ...
```

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

```
let printArray (arr : int array) =
  for elm in arr do
    printf "%d " elm
  printf "\n"

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

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

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

Listing 9.14: arrayCreation.fsx - Creating arrays with a syntax similarly to lists.

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

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

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

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

printArray A
square A
printArray A
```

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

Listing 9.15: arrayReassign.fsx - Arrays are mutable in spite the missing `mutable` keyword.

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

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

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

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

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

Listing 9.16: arrayPatternMatching.fsx - Only simple pattern matching is allowed for arrays.

The given example is the first example of a 2-dimensional array, which can be implemented as arrays of arrays and here written as `string array array`. Below further discussion of on 2 and higher dimensional arrays be discussed. Arrays support *slicing*, that is, indexing an array with a range results in a copy of array with values corresponding to the range, e.g., · slicing

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

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

Listing 9.17: arraySlicing.fsx - Examples of array slicing. Compare with Listing 9.8 and Listing 5.27.

As illustrated, the missing start or end index implies from the first or to the last element.
Arrays can be converted to lists and sequences by,

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

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

Listing 9.18: arrayConversion.fsx - The `Array` module contains functions for conversion to lists and sequences.

There are quite a number of built-in procedures for all arrays many which will be discussed in Chapter E.
Higher dimensional arrays can be created as arrays of arrays (of arrays . . .). These are known as *jagged arrays*, since there is no inherent control of that all sub-arrays are of similar size. E.g., the following is a jagged array of increasing width, · jagged arrays

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

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

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

Listing 9.19: arrayJagged.fsx - An array of arrays. When row lengths are of non-equal elements, then it is a Jagged array.

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

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

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

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

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

Listing 9.20: arrayJaggedSquare.fsx - A rectangular array.

Notice, the `for-in` cannot be used in `pownArray`, e.g., `for row in arr do for elm in row do elm <- pown elm p done done}` since the iterator value `\lstinline`elm! is not mutable even though `arr` is an array. In fact, square arrays of dimensions 2 to 4 are so common that F# has built-in modules for their support. In the following describe Array2D. The workings of Array3D and Array4D are very similar. An example of creating the same 2 dimensional array as above but as an `Array2D` is,

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

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

Listing 9.21: array2D.fsx - Creating a 3 by 4 rectangular arrays of intigers.

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

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

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

Listing 9.22: array2DSlicing.fsx - Examples of Array2D slicing. Compare with Listing 9.21.

Note that in almost all cases, slicing produces a sub rectangular 2 dimensional array except for `arr.[1,*]`, which is an array, as can be seen by the single `[`. In contrast, `A.[1..1,*]` is an Array2D. Note also, that `printfn` typesets 2 dimensional arrays as `[[ ... ]]` and not `[|[| ... |]|]`, which can cause confusion with lists of lists. [3]
Array2D and higher have a number of built-in functions that will be discussed in Chapter E.

9.4 Sequences

Sequences are lists, where the elements are build as needed. Examples are[4]

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

val a : seq<int>

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

val b : seq<int>

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

val c : seq<int>

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

val d : seq<float>
```

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

[4] **Mono does not support specification Spec-4.0 Section 6.3.11, seq comp-expr, in the form seq 3 or seq 3; 4.**

Listing 9.23: fsharpi, Creating sequences by range explicitly stating elements, a range expressions, a computation expression, and an infinite computation expression

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

```
range-exp = expr ".." expr [".." expr]
comp-expr =
  "let" pat "=" expr "in" comp-expr
  | "use" pat = expr "in" comp-expr
  | ("yield" | "yield!") expr
  | "if" expr "then" comp-expr ["else" comp-expr]
  | "match" expr "with" comp-rules
  | "try" comp-expr "with" comp-rules
  | "try" comp-expr "finally" expr
  | "while" expr "do" expr ["done"]
  | "for" ident "=" expr "to" expr "do" comp-expr ["done"]
  | "for" pat "in" expr-or-range-expr "do" comp-expr ["done"]
  | comp-expr ";" comp-expr
short-comp-expr = "for" pat "in" (expr | range-expr) "->" expr
comp-or-range-expr = comp-expr| short-comp-expr | range-expr
comp-rule = pat pattern-guardopt "->" comp-expr
comp-rules = comp-rule | comp-rule '|' comp-rules
expr = ...
  | "seq" "{" comp-or-range-expr "}" (* computation expression *)
  | ...
```

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

· yield bang

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

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

val sq : seq<int>

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

val itm : int = 1

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

val sbsq : seq<int>
```

Listing 9.24: fsharpi, Index a sequence with `Seq.item` and `Seq.take`

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

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

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

Listing 9.25: fsharpi, Sequences elements are first evaluated, when needed.

In the example, we see that the `printfn` function embedded in the definition is first executed, when the 3rd item is requested.
[5]

The only difference between computation expression's programming constructs and the similar regular expressions constructs is that they must return a value with the `yield` or `yield!` keywords.[6] The `try`-keyword constructions will be discussed in Chapter 10, and the `use`-keyword is a variant of `let` but used in asynchronous computations, which will not be treated here.
Infinite sequences is a useful concept in many programs and may be generated in a number of ways. E.g., to generate a repeated sequence, we could use recursive value definition, a computation expression, a recursive function, or the `Seq` module. Using a recursive value definition,

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

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

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

Listing 9.26: seqInfinteValue2.fsx - Recursive value definitions gives a warning. Compare with Listing9.27, 9.28, and 9.29.

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

Advice

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

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

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

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

Listing 9.27: seqInfinteValue.fsx - Infinite value definition without recursion nor warning. Compare with Listing 9.26, 9.28, and9.29.

or alternatively define a recursive function,

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

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

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

Listing 9.28: seqInfinteFunction.fsx - Recursive function definitions gives no a warning. Compare with Listing 9.26, 9.27, and 9.29.

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

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

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

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

Listing 9.29: seqInitInfinite.fsx - Using `Seq.initInfinte` and a function to generate an infinte sequence. Compare with Listing 9.26, 9.27, and 9.28.

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

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

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

Listing 9.30: seqConversion.fsx - Conversion between sequences and lists and arrays using the `List` module.

There are quite a number of built-in functions for sequences many which will be discussed in Chapter E. Lists and arrays may be created from sequences through the short-hand notation called *list and array sequence expressions*,

· list sequence expression

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

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

Part V

Appendix

Appendix A

Number systems on the computer

A.1 Binary numbers

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

- decimal number
- base
- decimal point
- digit

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

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

- bit
- binary

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

and examples are $1011.1_2 = 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 + 1 \cdot 2^{-1} = 11.5$. Notice that we use subscript 2 to denote a binary number, while no subscript is used for decimal numbers. The left-most bit is called the *most significant bit*, and the right-most bit is called the *least significant bit*. Due to typical organization of computer memory, 8 binary digits is called a *byte*, and 32 digits a *word*.
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.

- most significant bit
- least significant bit
- byte
- word
- 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,

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

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

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

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

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

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

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

I.e., the exponent is an integer, where $0 \leq e < 2^{11}$, and the mantissa is a rational, where $0 \leq m < 1$. For most combinations of e and m the real number v is calculated as,

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

with the exception that

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

· subnormals
· NaN
· not a number

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

$$e = 2^{11} - 2 = 2046 \tag{A.6}$$

$$m = \sum_{j=1}^{52} 2^{-j} = 1 - 2^{-52} \simeq 1. \tag{A.7}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Appendix B

Commonly used character sets

Letters, digits, symbols and space are the core of how we store data, write programs, and comunicate with computers and each others. These symbols are in short called characters, and represents a mapping between numbers, also known as codes, and a pictorial representation of the character. E.g., the ASCII code for the letter 'A' is 65. These mappings are for short called character sets, and due to differences in natural languages and symbols used across the globe, many different character sets are in use. E.g., the English alphabet contains the letters 'a' to 'z', which is shared by many other European languages, but which have other symbols and accents for example, Danish has further the letters 'æ', 'ø', and 'å'. Many non-european languages have completely different symbols, where Chinese character set is probably the most extreme, where some definitions contains 106,230 different characters albeit only 2,600 are included in the official Chinese language test at highest level.
Presently, the most common character set used is Unicode Transformation Format (UTF), whose most popular encoding schemes are 8-bit (UTF-8) and 16-bit (UTF-16). Many other character sets exists, and many of the later builds on the American Standard Code for Information Interchange (ASCII). The ISO-8859 codes were an intermediate set of character sets that are still in use, but which is greatly inferior to UTF. Here we will briefly give an overview of ASCII, ISO-8859-1 (Latin1), and UTF.

B.1 ASCII

The *American Standard Code for Information Interchange* (*ASCII*) [4], is a 7 bit code tuned for the letters of the english language, numbers, punctuation symbols, control codes and space, see Tables B.1 and B.2. The first 32 codes are reserved for non-printable control characters to control printers and similar devices or to provide meta-information. The meaning of each control characters is not universally agreed upon.

- American Standard Code for Information Interchange
- ASCII

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

- ASCIIbetical order

B.2 ISO/IEC 8859

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

- Latin1

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

Table B.1: ASCII

B.3 Unicode

Unicode is a character standard defined by the Unicode Consortium, `http://unicode.org` as the *Unicode Standard*. Unicode allows for 1,114,112 different codes. Each code is called a *code point*, which represents an abstract character. However, not all abstract characters requires a unit of several code points to be specified. Code points are divided into 17 planes each with $2^{16} = 65,536$ code points. Planes are further subdivided into named *blocks*. The first plane is called the *Basic Multilingual plane* and it are the first 128 code points is called the *Basic Latin block* and are identical to ASCII, see Table B.1, and code points 128-255 is called the *Latin-1 Supplement block*, and are identical to the upper range of ISO 8859-1, see Table B.3. Each code-point has a number of attributes such as the *unicode general category*. Presently more than 128,000 code points covering 135 modern and historic writing systems, and obtained at `http://www.unicode.org/Public/UNIDATA/UnicodeData.txt`, which includes the code point, name, and general category.

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

A unicode code point is an abstraction from the encoding and the graphical representation of a character. A code point is written as“U+” followed by its hexadecimal number, and for the Basic Multilingual plane 4 digits are used, e.g., the code point with the unique name LATIN CAPITAL LETTER A has the unicode code point is “U+0041”, and is in this text it is visualized as ’A’. More digits are used for code points of the remaining planes.

The general category is used in grammars to specify valid characters, e.g., in naming identifiers in F#. 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

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

Table B.2: ASCII symbols.

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

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

Code	Description
NBSP	Non-breakable space
SHY	Soft hypen

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

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 } ;
```

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

'`=`' definition, e.g.,

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

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

'`,`' concatenation, e.g.,

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

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

'`;`' termination of line

'`|`' alternative options, e.g.,

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

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

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

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

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

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

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

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

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

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

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

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

```
string = "abc"' ;
```

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

```
string = 'abc' ;
```

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

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

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

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

'-' exception, e.g.,

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

here **consonant** are all letters except vowels.

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

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

```
string = character  { character }
terminal = "'"  string  "'" | '"'  string  '"'
rhs = identifier
  | terminal
  | "["  rhs  "]"
  | "{"  rhs  "}"
  | "("  rhs  ")"
  | rhs  "|"  rhs
(*  | rhs  ","  rhs  *)
rule = identifier  "="  rhs  (* ";" *)
grammar = rule { rule }
```

Here the comments demonstrate, the relaxed modification.

Appendix D

Language Details

[1]

[1] **Somewhere I should possibly talk about Lightweight Syntax, Spec-4.0 Chapter 15.1**

Operator	Associativity	Description
`ident "<"types ">"`	Left	High-precedence type application
`ident "("expr ")"`	Left	High-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 D.1: Precedence and associativity of operators. Operators in the same row has same precedence. See Listing 6.28 for the definition of `prefixOp`

Bibliography

[1] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[2] Programming Research Group. Specifications for the ibm mathematical formula translating system, fortran. Technical report, Applied Science Division, International Business Machines Corporation, 1954.

[3] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[4] X3: ASA Sectional Committee on Computers and Information Processing. American standard code for information interchange. Technical Report ASA X3.4-1963, American Standards Association (ASA), 1963. `http://worldpowersystems.com/projects/codes/X3.4-1963/`.

[5] George Pólya. *How to solve it.* Princeton University Press, 1945.

Index