Learning to program with F#

Jon Sporring

July 18, 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*

`.fscript` 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 `";;"` token, or to execute a script file treated as a single script-fragment. To illustrate the difference, consider the following program, which declares a value `a` to be the decimal value 3.0 and finally print it to the console:

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

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

```
$ fsharpi

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

For help type #help;;

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

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

> #quit;;
```

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

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

Finally, compiling and executing the code is performed as,

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

Both the interpreter and the compiler translates the source code into a format, which can be executed by the computer. While the compiler performs this translation once and stores the result in the executable file, the interpreter translates the code every time the code is executed. Thus, to run the program again with the interpreter, then it must be retranslated as "`$fsharpi gettingStartedStump.fsx`", but since the program has been compiled, then the compile-execute only needs the be re-executed "`$ 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`, $1.88\text{s} \gg 0.05\text{s}$.
[1]

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

· type inference
· debugging
· unit-testing

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

Chapter 4

Quick-start guide

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

> What is the sum of 357 and 864?

we have written the following program in F#,

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

```
1221
```

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

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

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

- 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

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 adviced 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 arithmatic 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 arithmatic.

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

```
> let a = 357;;

val a : int = 357

> let b = 863.4;;

val b : float = 863.4

> let c = a + b;;

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

```
/Users/sporring/Desktop/fsharpNotes/stdin(3,13): error FS0001: The type '
    float' does not match the type 'int'
```
Listing 4.4: fsharpi

we see that binding a name to a number without a decimal point is inferred to be integer, while when binding to a number with a decimal point, then the type is inferred to be a float, and when trying to add values of integer and floating point, then we get an error.
F# is a functional first programming language, and one implication is that names are constant and cannot be changed. If attempted, then F# will return an error as, e.g.,

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

```
/Users/sporring/repositories/fsharpNotes/quickStartRebindError.fsx(2,5):
    error FS0037: Duplicate definition of value 'a'
```
Listing 4.5: quickStartRebindError.fsx - A name cannot be rebound.

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

```
> let a = 357;;

val a : int = 357

> let a = 864;;

val a : int = 864
```
Listing 4.6: fsharpi

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

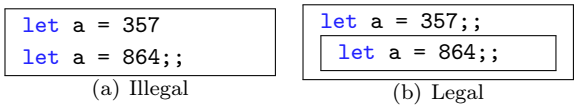
- ;;
- token
- scope
- nested scope
- overshadow

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

  let a = 864;;
  ----^

/Users/sporring/Desktop/fsharpNotes/stdin(2,5): error FS0037: Duplicate
    definition of value 'a'
```
Listing 4.7: fsharpi

Scopes can be visualized as nested squares as shown in Figure 4.1.
In F# *functions* are also values, and defining a function `sum` as part of the solution to the above program gives,

- function

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



(a) Illegal (b) Legal

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

```
1221
```

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

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

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

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

> let c = sum 357.6 863.4;;

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

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

Listing 4.9: fsharpi

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

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

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

Listing 4.10: fsharpi

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

Chapter 5

Numbers, Characters, and Strings

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

· type
· primitive types
· literal
· literal type

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

That is, the *let* keyword indicates that the following is a binding of an identifier with an expression, and that the type may be specified with the *:* token. An identifier must start with a letter, but can be followed by zero or more of letters, digits, and a range of special characters. For characters in the Basic Latin Block, i.e., the first 128 code points alias ASCII characters, an ident is,

· let
· :

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

Thus, examples of identifiers are `a`, `theCharacter9`, `Next_Word`, `_tok`. The for the full definition, `letter = Lu | Ll | Lt | Lm | Lo | Nl` and `special-char = 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 `special-char` category. The binding may be mutable, which will be discussed in Section 5.4, and the binding may only be for the last expression as indicated by the *in* keyword. The simplest example of an expression is a *literal*, i.e., a constant such as the number 3. Examples of let statements with *literals* are

· in
· literal
· literals

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

val a : int = 3
val b : uint32 = 4u
val c : float = 5.6
val d : float32 = 7.9000001f
```

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

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

```
val e : char = 'A'
val f : byte = 66uy
val g : string = "ABC"
val h : unit = ()
```
Listing 5.1: fsharpi, binding identifiers and literals.

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

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

val a : int = 3
val b : float = 3.0
val c : char = '3'
val d : string = "3"
```
Listing 5.2: fsharpi, many representations of the number 3 but using different types.

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

```
  > let a : float = 3;;

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

/Users/sporring/repositories/fsharpNotes/stdin(50,17): error FS0001: This
    expression was expected to have type
    float
but here has type
    int
```
Listing 5.3: fsharpi, binding error due to type mismatch.

since the left-hand-side is an identifier of type float, while the right-hand-side is a literal of type integer. Many primitive types are compatible and the type of a literal may be changed by *type casting*. E.g., · type casting

```
> let a = float 3;;

val a : float = 3.0
```
Listing 5.4: fsharpi, casting an integer to a floating point number.

where the left-hand-side is inferred to by of type float, since the integer number `3` is casted to float resulting in a similar floating point value, in this case the float point number `3.0`. As a technical detail, `float` is here a function rather than a type, which takes the argument `3` and returns the value `3.0`. For more on functions see Section 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,

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

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

Listing 5.5: fsharpi, 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 part after the decimal point without rounding,

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

```
357.6 -> 357
```

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

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

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

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

```
357.6 -> 358
```

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

since if $y.x \in [y.0, y.5)$, then $y.x + 0.5 \in [y.5, y+1)$, from which downcasting removes the fractional part resulting in y. And if $y.x \in [y.5, y+1)$, then $y.x + 0.5 \in [y+1, y+1.5)$, from which downcasting removes the fractional part resulting in $y+1$. Hence, the result is rounding.
If parentheses are omitted in Listing 5.7, then F# will interpret the expression as `(int a)+ 0.5`, which is erroneous, since addition of an integer with a float is undefined. This is an example of precedence, i.e., function evaluation takes precedence over addition meaning that it is performed before addition. Consider the arithmetic expression, whose result is bound to `a` by

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

val a : int = 23
```

Listing 5.8: fsharpi, a simple arithmetic expression.

Here, the addition and multiplication functions are shown in *infix notation* with the *operator* tokens `+` and `*`. To arrive at the resulting value 23, F# has to decide in which order to perform the calculation.

· infix notation
· operator

Operator	Associativity	Example	Description
`+op, -op, ~~~op`	Left	`-3`	Unary identity, negation, and bitwise negation operator
`f x`	Left	`f 3`	Function application
`op ** op`	Right	`3.0 ** 2.0`	Exponent
`op * op, op / op, op % op`	Left	`3.0 / 2.0`	Multiplication, division and remainder
`op + op, op - op`	Left	`3.0 + 2.0`	Addition and subtraction binary operators
`op ^^^ op`	Right	`0xAAuy ^^^ 0xFFuy`	bitwise exclucive or
`op < op, op <= op, op > op, op >= op, op = op, op <> op, op <<< op, op >>> op, op &&& op, op ||| op,`	Left	`3 > 5`	Comparison operators, bitwise shift, and bitwise 'and' and 'or'.
`&&`	Left	`true && true`	Boolean and
`||`	Left	`true || true`	Boolean or

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

There are 2 possible orders, `3 + (4 * 5)` or `(3 + 4)* 5`, which gives different results. For integer arithmetic, the correct order is of course to multiply before addition, and we say that multiplication takes *precedence* over addition. Every atomic operation that F# can perform is ordered in terms of its precedences, and for some common built-in operators shown in Table 5.2, the precedence is shown by the order they are given in the table. Associativity implies the order in which calculations are performed for operators of same precedence. For some operators and type combinations association matters little, e.g., multiplication associates to the left and exponentiation associates to the right, e.g., in[1]

· precedence

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

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

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

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

Listing 5.9: fsharpi, precedences rules define implicite parantheses.

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

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

[1] **Spec-4.0, Table 18.2.1 appears to be missing boolean 'and' and 'or' operations. Section 4.4 seems to**

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

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

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

```
9.0
```

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

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

· scope

5.1 Booleans

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

· and
· or
· not
· truth table

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

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

Listing 5.11: fsharpi, boolean operators and truth tables.

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

be missing &&&& and ||| bitwise operators.

5.2 Integers and Reals

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

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

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

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

Listing 5.12: fsharpi, unary integer negation operator.

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

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

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

Listing 5.13: fsharpi, binary integer addition, subtraction, and multiplication operators.

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

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

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

Listing 5.14: fsharpi, binary integer division and remainder operators.

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

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

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

```
val y : int = 7
```
Listing 5.15: fsharpi, binary division and remainder is a lossless representation of an integer.

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

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

```
> pown 2 5;;
val it : int = 32
```
Listing 5.16: fsharpi, integer exponentiation function, and the irrelevant identifier.

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

· `it`

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

val it : int = 32
```
Listing 5.17: fsharpi, the equivalent to the irrelevant identifier.

Rumor has it, that the identifier `it` is an abbreviation for 'irrelevant'.
Performing arithmetic operations on `int` types requires extra care, since the result may cause *overflow*, *underflow*, and even exceptions, e.g., the range of the integer type `sbyte` is $[-128 \ldots 127]$, which causes problems in the following example,

· overflow
· underflow

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

val a : sbyte = 100y
val b : sbyte = 30y
val c : sbyte = -126y
```
Listing 5.18: 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`,

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

val a : sbyte = -100y
val b : sbyte = -30y
val c : sbyte = 126y
```
Listing 5.19: fsharpi, subtracting integers may cause underflow

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$FSI_0007>.$FSI_0007.main@ () <0x6b78180 + 0x0000e> in <
      filename unknown>:0
  at (wrapper managed-to-native) System.Reflection.MonoMethod:
```

```
      InternalInvoke (System.Reflection.MonoMethod,object,object[],System.
      Exception&)
  at System.Reflection.MonoMethod.Invoke (System.Object obj, BindingFlags
      invokeAttr, System.Reflection.Binder binder, System.Object[]
      parameters, System.Globalization.CultureInfo culture) <0x1a55ba0 + 0
      x000a1> in <filename unknown>:0
Stopped due to error
```

Listing 5.20: 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 9 · run-time error

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

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

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

Listing 5.21: fsharpi, integer types may be specified as binary, octal, and hexadecimal numbers.

For a description of binary representations see Appendix A.1. The overflow error in Listing 5.18 can be understood in terms of the binary representation of integers: In binary, $130 = 10000010_2$, and this binary pattern is interpreted differently as `byte` and `sbyte`,

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

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

Listing 5.22: 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`.
For binary arithmatic on integers, the following operators are available:

`op <<< n`**:** Bitwise left shift, shifts any integer bit pattern `n` positions to the left insert 0's to right.

`op >>> n`**:** Bitwise left right, shifts any integer bit pattern `n` 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

`op1 ~~~ op1`**:** Bitwise xor, which is returns the result of the boolean 'xor' operator defined by,

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

position-wise.

Unfortunately, there are no built-ind functions outputting integers on binary form, so to understand the output of the following program,

```
> let a = 0b11000011uy
- let b = a <<< 1
- let c = a >>> 1
- let d = ~~~a
- let e = a ^^^0b11111111uy;;

val a : byte = 195uy
val b : byte = 134uy
val c : byte = 97uy
val d : byte = 60uy
val e : byte = 60uy
```

Listing 5.23: fsharpi, the left most bit is interpreted differently for signed and unsigned integers, which gives rise to potential overflow errors.

we must consider the 8-bit binary form of the unsigned integers: $195 = 11000011_2$, $134 = 10000110_2$, $97 = 01100001_2$, and $60 = 00111100_2$, which agrees with the definitions. [2]
For floating point numbers the following arithmetic operators are defined:

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

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

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

Listing 5.24: fsharpi, unary floating point negation operator.

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

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

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

Listing 5.25: fsharpi, binary floating point addition, subtraction, multiplication, and division operators.

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

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

[2] **mention somewhere that comparison operators will be treated later.**

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

Listing 5.26: fsharpi, binary floating point division and remainder operators.

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

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

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

Listing 5.27: fsharpi, floating point division, truncation, and remainder is a lossless representation of a number.

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

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

```
> let a = 2.0 ** 5.0;;

val a : float = 32.0
```

Listing 5.28: fsharpi, binary floating point exponentiation.

which is equal to 2^5.

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

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

val a : float = infinity
val b : float = nan
```

Listing 5.29: 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.,

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

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

Listing 5.30: fsharpi, floating point arithmatic has finite precision.

Hence, although `c` appears to be correctly calculated, by the subtraction we see, that the value bound in `c` is not exactly the same as `358.0`, and the reason is that the neither `357.8` nor `0.1` are exactly representable as a `float`, which is why the repeated addition accumulates a small representation error. F# allows for assigning *unit of measure* to `float`, `float32`, `decimal` and signed integer such as `int` . · unit of measure

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

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

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

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

Listing 5.31: fsharpi, floating point and integer numbers may be assigned units of measure.

However, if we mixup units of measure under addition, then we get an error,

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

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

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

Listing 5.32: fsharpi, units of measure adds an extra layer of types for syntax checking at compile time.

Units of measure allow for `*`, `/`, and `^`[3] for multiplication, division and exponentiation. Values with units can be casted to *unit-less* values by casting, and back again by multiplication as, · unit-less

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

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

[3] **Spec-4.0: this notation is inconsistent with ** for float exponentiation.**

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

Table 5.4: International System of Units.

```
val b : int = 2
val c : int<m> = 2
```

Listing 5.33: fsharpi, type casting units of measure.

Compound symbols can be declared as,

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

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

Listing 5.34: fsharpi, aggregated units of measure.

For fans of the metric system there is the International System of Units, and these are built-in in `Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols` and give in Table 5.4. Hence, using the predefined unit of seconds, we may write,

Character	Escape sequence	Description
BS	`\b`	Backspace
LF	`\n`	Newline
CF	`\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 5.5: Escape characters. For the unicode characters 'X' are hexadecimal digits, while for tricode characters 'D' is a decimal character.

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

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

Listing 5.35: fsharpi, SI units of measure are built-in.

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

· open

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

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

Listing 5.36: fsharpi, simpler syntax by importing, but beware of namespace polution.

The open keyword should be used with care, since now all the bindings in `Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols` have been imported into the present scope, and since we most likely do not know, which bindings have been used by the programmers of `Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols`, we do not know which identifiers to avoid, when using let statements. We have obtained, what is known as *namespace pollution*. Read more about namespaces in Part IV.

· namespace pollution

Using units of measure is advisable for calculations involving real-world values, since some semantical errors of arithmetic expressions may be discovered by checking the resulting unit of measure.
[4]

5.3 Chars and Strings

A character is a Unicode code point, see Appendix B.3 for a description of code points, and character literals enclosed in single quotation marks,[5]

```
    char = "'" charOrEscape "'"
```

where `charOrEscape` are code points or escape sequence starting with \ as illustrated in Table 5.5. Examples are `'a'`, `'_'`, `'\n'`. The trigraph `\DDD` uses decimal specification for the first 256 unicode characters. The hexadecimal escape codes `\uXXXX`, `\UXXXXXXXX` allow for the full specification of any unicode character.

[4] **add comparsion operators!**
[5] **Spec-4.0 p.28:** `char-char` **is missing option** `unicodegraph-long`

Character arithmatic is most often done by in integer space. 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 5.37: fsharpi, converting case by casting and integer arithmatic.

A string is a sequence of characters enclosed in double quotation marks,[6]

```
    string-expr = '"' { char | LF | SP }  '"'
```

Examples are `"a"`, `"this is a string"`, and `"-&#\@"`. Newlines and following whitespaces are taken literal, but may be ignored by a preceding \character, and strings literals may be *verbatim* by preceding the string with '@', meaning that the escape sequences are not converted to their code point, e.g.,

· verbatim

```
> let a = "abc
- de"
- let b = "abc\
-   de"
- let c = "abc\nde"
- let d = "abcde"
- let e = @"abc\nde";;

val a : string = "abc
de"
val b : string = "abcde"
val c : string = "abc
de"
val d : string = "abcde"
val e : string = "abc\nde"
```

Listing 5.38: fsharpi, examples of string literals.

The response is shown in double quotation marks, which are not part of the string. Verbatim literals containing double quotation marks are escaped with an extra double quotation mark, or the alternative tripple double quotation mark may be used, e.g.,

```
> let a = @"This is a verbatim ""quote""."
- let b = """This is a verbatim "quote"."""";;

val a : string = "This is a verbatim "quote"."
val b : string = "This is a verbatim "quote"."
```

Listing 5.39: fsharpi, example of double quotation marks in verbatim string literals.

Operations on `string` is quite rich. The most simple is concatenation using `+` token, e.g.,

```
> let a = "hello"
- let b = "world"
- let c = a + " " + b;;

val a : string = "hello"
val b : string = "world"
```

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

```
val c : string = "hello world"
```
Listing 5.40: 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,

```
> let a = "abcdefg"
- let b = a.[0]
- let c = a.[3]
- ;;

val a : string = "abcdefg"
val b : char = 'a'
val c : char = 'd'
```
Listing 5.41: fsharpi, example of string indexing.

The *dot notation* is an example of Structured programming, where technically `a` is an immutable *object* of *class* `string`, and `[]` is an object *method*. For more on object, classes, and methods see Chapter 16. Notice, that the first character has index 0, and to get the last character in a string, we use the string's lenght property as,

· dot notation
· object
· class
· method

```
> let a = "abcdefg"
- let l = a.Length
- let first = a.[0]
- let last = a.[l-1];;

val a : string = "abcdefg"
val l : int = 7
val first : char = 'a'
val last : char = 'g'
```
Listing 5.42: 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 `a.[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 13.1.

5.4 Mutable bindings

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

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

Mutable data is synonymous with the term *variable*. A variable is an area in the 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 5.43: 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 `<-` token. The `<-` token 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 5.44: fsharpi, example of changing the content of a variable.

then we instead would have obtained the default assignment of the result of the comparision 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 |<-| operator.
Assignment type mismatches will result in an error,

```
let mutable x = 5
printfn "%d" x
x <- -3.0
printfn "%d" x
```

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

Listing 5.45: mutableAssignReassingTypeError.fsx - Assignmetn type mismatching causes a compile time error.

I.e., once the type of an identifier has been declared or infered, 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 5.46: mutableAssignIncrement.fsx - Variable increment is a common use of variables.

which is an example we will return to many times later in this text.

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

- byte
- word

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

- octal
- hexadecimal

A.2 IEEE 754 floating point standard

The set of real numbers also called *reals* includes all fractions and irrational numbers. It is infinite in size both in the sense that there is no largest nor smallest number and between any 2 given numbers there are infinitely many numbers. Reals are widely used for calculation, but since any computer only has finite memory, it is impossible to represent all possible reals. Hence, any computation performed on a computer with reals must rely on approximations. *IEEE 754 double precision floating-point format* (*binary64*), known as a *double*, is a standard for representing an approximation of reals using 64 bits. These bits are divided into 3 parts: sign, exponent and fraction,

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

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

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

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

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

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

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

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

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

with the exception that

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

· subnormals
· NaN
· not a number

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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