Learning to program with F#

Jon Sporring

September 6, 2016

Contents

Chapter 4

Quick-start guide

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

> **Problem:**
>
> 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 2.1) 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,

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*. 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
```

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

```
let a = 357
let a = 864;;
```
(a) Invalid

```
let a = 357;;
  let a = 864;;
```
(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.

```
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/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.

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,

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

```
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
	int32	Synonymous with int
	uint32	Integer values from 0 to 4,294,967,295
Real	**float**	64-bit IEEE 754 floating point value from $-\infty$ to ∞
	double	Synonymous with float
Character	**char**	Unicode character
	string	Unicode sequence of characters
None	**unit**	No value denoted
Object	**obj**	An object
Exception	**exn**	An exception

Table 5.1: List of some of the basic types. The most commonly used types are highlighted in bold. For at description of integer see Appendix 1.1, for floating point numbers see Appendix 1.2, for ASCII and Unicode characters see Appendix 2, for objects see Chapter 20, and for exceptions see Chapter 11.

```
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 = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7";
xDigit =
  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
  | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" | "f";
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`.

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

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

A *character* is a *Unicode code point*, and character literals are enclosed in single quotation marks, see Appendix 2.3 for a description of code points. The EBNF for characters is,

· character
· Unicode
· code point

```
codePoint = ?Any unicode codepoint?;
escapeChar =
  "\" ("b" | "n" | "r" | "t" | "\" | '"' | "'" | "a" | "f" | "v")
  | "\u" xDigit xDigit xDigit xDigit
  | "\U" xDigit xDigit xDigit xDigit xDigit xDigit xDigit xDigit
  | "\" dDigit dDigit dDigit;
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,

· string

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

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

· newline
· whitespace

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

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.

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

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

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

Listing 5.4: fsharpi, Named and implied literals.

Strings literals may be *verbatim* by the @-notationmeaning 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.

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,

· expression
· infix operator

```
const =
  byte
  | sbyte
  | int32
  | uint32
  | int
  | ieee64
  | char
  | string
  | verbatimString
  | "false"
  | "true"
  | "()";
sliceRange =
  expr
  | expr ".." (*no space between expr and ".."*)
  | ".." expr (*no space between expr and ".."*)
  | expr ".." expr (*no space between expr and ".."*)
  | "*";
expr = ...
  | const (*a const value*)
  | "(" expr ")" (*block*)
  | expr expr (*application*)
```

<table>
<tr><th>Operator</th><th>op1</th><th>op2</th><th>Expression</th><th>Result</th><th>Description</th></tr>
<tr><td>op1 + op2</td><td>ints
floats
chars
strings</td><td>ints
floats
chars
strings</td><td>5 + 2
5.0 + 2.0
'a' + 'b'
"ab" + "cd"</td><td>7
7.0
'\195'
"abcd"</td><td>Addition

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

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.

```
  | expr infixOp expr (*infix application*)
  | prefixOp expr (*prefix application*)
  | expr ".[" expr "]" (*index lookup, no space before "."*)
  | expr ".[" sliceRange "]" (*index lookup, no space before "."*)
```

Listing 5.10: expressionArithmetic

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. [1]

· 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

[1] Todo: **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

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

Table 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 E.1.

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.

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

· 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;;
```

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.

```
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.

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 1. 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 $[-128\ldots127]$, 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 11 · 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 1. 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 F.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 20.
· dot notation
· object
· class
· method
Strings are compared letter by letter. For two strings to be equal, they must have the same length and all the letters must be identical. E.g., `"abs" = "absalon"` is false, while `"abs" = "abs"` is true. The `<>` operator is the boolean negation of the `=` operator, e.g., `"abs" <> "absalon"` is true, while `"abs" <> "abs"` is false. For the `<` , `<=`, `>`, and `>=` operators, the strings are ordered alphabetically, such that `"abs" < "absalon"&& "absalon"< "milk"` is true, that is, the `<` operator on two strings is true, if the left operand should come before the right, when sorting alphabetically. The algorithm for deciding the boolean value of `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 1

Number systems on the computer

1.1 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

$$v = \sum_{i=-m}^{n} d_i 10^i \tag{1.1}$$

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

$$v = \sum_{i=-m}^{n} b_i 2^i \tag{1.2}$$

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 1.1.

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

1.2 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 1.1: 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,

$$e = \sum_{i=1}^{11} e_i 2^{11-i}, \tag{1.3}$$

$$m = \sum_{j=1}^{52} m_j 2^{-j}. \tag{1.4}$$

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,

$$v = (-1)^s (1 + m) 2^{e-1023} \tag{1.5}$$

with the exception that

	$m = 0$	$m \neq 0$
$e = 0$	$v = (-1)^s 0$ (signed zero)	$v = (-1)^s m2^{1-1023}$ (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

$$e = 2^{11} - 2 = 2046 \tag{1.6}$$

$$m = \sum_{j=1}^{52} 2^{-j} = 1 - 2^{-52} \simeq 1. \tag{1.7}$$

$$v_{\max} = \pm \left(2 - 2^{-52}\right) 2^{1023} \simeq \pm 2^{1024} \simeq \pm 10^{308} \tag{1.8}$$

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

$$v = (-1)^s \left(1 + \sum_{j=1}^{52} m_j 2^{-j}\right) 2^{52} \tag{1.9}$$

$$= \pm \left(2^{52} + \sum_{j=1}^{52} m_j 2^{-j} 2^{52}\right) \tag{1.10}$$

$$= \pm \left(2^{52} + \sum_{j=1}^{52} m_j 2^{52-j}\right) \tag{1.11}$$

$$\overset{k=52-j}{=} \pm \left(2^{52} + \sum_{k=51}^{0} m_{52-k} 2^k\right) \tag{1.12}$$

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

$$v \overset{k=53-j}{=} \pm \left(2^{53} + \sum_{k=52}^{1} m_{53-k} 2^k\right) \tag{1.13}$$

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,

$$v \overset{k=51-j}{=} \pm \left(2^{51} + \sum_{k=50}^{-1} m_{51-k} 2^k\right) \tag{1.14}$$

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

$$
\begin{aligned}
v &= (-1)^s \left(\sum_{j=1}^{52} m_j 2^{-j} \right) 2^{-1022} && (1.15)\\
&= \pm \left(\sum_{j=1}^{52} m_j 2^{-j} 2^{-1022} \right) && (1.16)\\
&= \pm \left(\sum_{j=1}^{52} m_j 2^{-j-1022} \right) && (1.17)\\
&\overset{k=-j-1022}{=} \pm \left(\sum_{j=-1023}^{-1074} m_{-k-1022} 2^{k} \right) && (1.18)
\end{aligned}
$$

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

Chapter 2

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.

2.1 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 2.1 and 2.2. The first 32 codes are reserved for non-printable control characters to control printers and similar devices or to provide meta-information. The meaning of each control characters is not universally agreed upon.
The code order is known as *ASCIIbetical order*, and it is sometimes used to perform arithmetic on codes, e.g., an upper case letter with code c may be converted to lower case by adding 32 to its code. The ASCIIbetical order also has consequence for sorting, i.e., when sorting characters according to their ASCII code, then ’A’ comes before ’a’, which comes before the symbol ’{’.

· American Standard Code for Information Interchange
· ASCII
· ASCIIbetical order

2.2 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 2.3 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 2.1: ASCII

2.3 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 2.1, 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 2.3. 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 2.5.

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 2.2: 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 2.3: 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 2.4: 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 2.5: Some general categories for the first 256 code points.

Chapter 3

A brief introduction to Extended Backus-Naur Form

Extended Backus-Naur Form (*EBNF*) is a language to specify programming languages in. The name is a tribute to John Backus who used it to describe the syntax of ALGOL58 and Peter Nauer for his work on ALGOL 60.

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

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

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

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

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

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

'=' definition, e.g.,

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

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

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

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

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

'`;`' termination of line

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
codepoint = ?Any unicode codepoint?;
```

'`-`' exception, e.g.,

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

here `consonant` are all letters except vowels.

Rules for rewriting EBNF are:

<table>
<tr><th>Rule</th><th>Description</th></tr>
<tr><td><code>s | t</code> ↔ <code>t | s</code></td><td><code>|</code> is commutative</td></tr>
<tr><td><code>r | (s | t)</code> ↔ <code>(r | s)| t</code> ↔ <code>r | s | t</code></td><td><code>|</code> is associative</td></tr>
<tr><td><code>(r s)t</code> ↔ <code>r (s t)</code> ↔ <code>r s t</code></td><td>concatenation is associative</td></tr>
<tr><td><code>r (s | t)</code> ↔ <code>r t| r s</code>
<code>(r | s)t</code> ↔ <code>r t| r t</code></td><td>concatenation is distributive over <code>|</code></td></tr>
<tr><td><code>[s | t]</code> ↔ <code>[t] | [s]</code></td><td></td></tr>
<tr><td><code>[[s]]</code> ↔ <code>[s]</code></td><td><code>[]</code> is idempotent</td></tr>
<tr><td><code>{{s}}</code> ↔ <code>{s}</code></td><td><code>{}</code> is idempotent</td></tr>
</table>

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

Symbol	Description
$\star$	repetition
$-$	except
,	concatenate
|	option
=	define
;	terminator

in order of precedence, such that $\star$ has higher precedence than $-$. These precedence rules are overridden by bracket pairs, such as `' '`, `" "`, `(* *)`, `( )`, `[ ]`, `{ }`, `? ?`.
The proposal allows for identifies that includes space, but often a reduced form is used, where identifiers are single words, in which case the concatenation symbol `,` is replaced by a space. Likewise, the termination symbol `;` is often replaced with the new-line character, and if long lines must be broken, then indentation is used to signify continuation. In this relaxed EBNF, the EBNF syntax itself can be expressed in EBNF as,

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

Here the comments demonstrate, the relaxed modification. Newline does not have an explicit representation in EBNF, which is why we use `? ?` brackets

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.