

Learning to program with F#

Jon Spurring

September 8, 2016

Contents

1	Preface	4
2	Introduction	5
2.1	How to learn to program	5
2.2	How to solve problems	6
2.3	Approaches to programming	6
2.4	Why use F#	7
2.5	How to read this book	8
I	F# basics	9
3	Executing F# code	10
3.1	Source code	10
3.2	Executing programs	10
4	Quick-start guide	13
5	Using F# as a calculator	17
5.1	Literals and basic types	17
5.2	Operators on basic types	22
5.3	Boolean arithmetic	24
5.4	Integer arithmetic	25
5.5	Floating point arithmetic	27
5.6	Char and string arithmetic	28

6	Constants, functions, and variables	34
6.1	Values	36
6.2	Non-recursive functions	40
6.3	User-defined operators	43
6.4	The Printf function	45
6.5	Variables	47
7	In-code documentation	52
8	Controlling program flow	57
8.1	For and while loops	57
8.2	Conditional expressions	61
8.3	Programming intermezzo	62
8.4	Recursive functions	63
9	Ordered series of data	65
9.1	Tuples	66
9.2	Lists	68
9.3	Arrays	71
10	Testing programs	76
10.1	White-box testing	78
10.2	Back-box testing	80
10.3	Debugging by tracing	83
11	Exceptions	91
12	Input and Output	97
12.1	Interacting with the console	97
12.2	Storing and retrieving data from a file	98
12.3	Working with files and directories.	102
12.4	Programming intermezzo	103

II	Imperative programming	105
13	Graphical User Interfaces	107
14	Imperative programming	108
14.1	Introduction	108
14.2	Generating random texts	110
14.2.1	0'th order statistics	110
14.2.2	1'th order statistics	112
III	Declarative programming	113
15	Sequences and computation expressions	114
15.1	Sequences	114
16	Patterns	119
16.1	Pattern matching	119
17	Types and measures	121
17.1	Unit of Measure	121
18	Functional programming	124
IV	Structured programming	127
19	Namespaces and Modules	128
20	Object-oriented programming	129
V	Appendix	130
A	Number systems on the computer	131
A.1	Binary numbers	131
A.2	IEEE 754 floating point standard	131

B	Commonly used character sets	135
B.1	ASCII	135
B.2	ISO/IEC 8859	135
B.3	Unicode	136
C	A brief introduction to Extended Backus-Naur Form	139
D	F_b	142
E	Language Details	147
E.1	Precedence and associativity	147
E.2	Behind the scene	147
E.3	Lightweight Syntax	147
F	The Some Basic Libraries	149
F.1	System.String	149
F.2	List, arrays, and sequences	154
F.3	Mutable Collections	156
F.3.1	Mutable lists	156
F.3.2	Stacks	156
F.3.3	Queues	157
F.3.4	Sets and dictionaries	157
	Bibliography	158
	Index	159

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,

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

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

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

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

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

Each literal represent the number 3, but their types are different, and hence they are quite different values. The types *int* for integer numbers, *float* for floating point numbers, *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*

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

Table 5.1: List of some of the basic types. The most commonly used types are highlighted in bold. For at description of integer see Appendix A.1, for floating point numbers see Appendix A.2, for ASCII and Unicode characters see Appendix B, for objects see Chapter 20, and for exceptions see Chapter 11.

10 means that for a number consisting of a sequence of digits separated by a *decimal point*, where each *digit* can have values $d \in \{0, 1, 2, \dots, 9\}$, and the value, which each digit represents is proportional to its position. The part before the decimal point is called the *whole part* and the part after is called the *fractional part* of the number. The whole part without a decimal point and a fractional part is called an *integer 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,

Listing 5.3: Decimal numbers.

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

meaning that a dDigit is either 0 or 1 or ... or 9, an dInt is one or more dDigit such as 3, 45, and 0124972930485738. A dFloat is one or more digits, a dot and zero or more digits, such as 0.4235, 3., but not .5 nor .. There is no space between the digits and between digits and the dot.

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

Listing 5.4: Scientific notation.

```
sFloat = (dInt | dFloat) ("e" | "E" ) ["+" | "-"] dInt; (*no spaces*)
float = dFloat | sFloat;
```

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

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

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

- scientific notation

- bit
- binary number
- octal number
- hexadecimal number

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.

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,

Listing 5.5: Binary, heximal, and octal numbers.

```

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

```

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

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

- character
- Unicode
- code point

Listing 5.6: Character escape sequences.

```

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

```

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

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

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

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

· string

Listing 5.7: Strings.

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

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

· newline
· whitespace

Listing 5.8: Whitespace and newline.

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

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

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

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

The response is shown in double quotation marks, which are not part of the string.

F# supports *literal types*, where the type of a literal is indicated as a prefix or suffix as shown in the Table 5.3. Examples are,

· literal type

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

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

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

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

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

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

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

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

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

Listing 5.13, castingBooleans.fsx:
Casting booleans.

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

Here `System.Convert.ToBoolean` is the identifier of a function `ToBoolean`, which is a *member* of the *class* `Convert` that is included in the *namespace* `System`. Namespaces, classes, and members are all

· verbatim

· typecasting

· member
· class
· namespace

part of Structured programming to be discussed in Part IV.

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

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

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

Here we typecasted to a lesser type, in the sense that integers is a subset of floating point numbers, and this is called *downcasting*. The opposite is called *upcasting* and is often non-destructive, as Listing 5.12 showed, where an integer was casted to a float while retaining its value. As a side note, *rounding* a number $y.x$, where y is the *whole part* and x is the *fractional part*, is the operation of mapping numbers in the interval $y.x \in [y.0, y.5)$ to y and $y.x \in [y.5, y+1)$ to $y+1$. This can be performed by downcasting as follows,

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

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

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

5.2 Operators on basic types

Listing 5.15 is an example of an arithmetic *expression* using an *infix operator*. Expressions is the basic building block of all F# programs, and its grammar has many possible options. The grammar for expressions are defined recursively, and some of it is given by,

Listing 5.16: Expressions.

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

Recursion means that a rule or a function is used by the rule or function itself in its definition. 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 positive integer. Since the operator appears before the operand it is a *prefix operator*, and since it only takes one argument it is also a *unary operator*. Finally, some expressions are function names, which can be applied to expressions. F# supports a range of arithmetic infix and prefix operators on its built-in types such as addition, subtraction, multiplication, division, and exponentiation using the `+`, `-`, `*`, `/`, `**` binary operators respectively. Not all operators are defined for all types, e.g., addition is defined for integer and float types as well as for characters and strings, but multiplication is only defined for integer and floating point types. A complete list of built-in operators on basic types is shown in Table ?? and ?? and a range of mathematical functions shown in Table ??.

- operator
- operands
- binary operator
- prefix operator
- unary operator

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

- precedence

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

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

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

- infix notation
- operator
- precedence

Associativity implies the order in which calculations are performed for operators of same precedence. For some operators and type combinations association matters little, e.g., multiplication associates to the left and exponentiation associates to the right, e.g., in

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

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

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

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

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

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 5.4. 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

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

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

val it : unit = ()
```

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 lightweight syntax discussed in Chapter 6.

5.4 Integer arithmetic

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

Table ??, ??, and ?? 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 $[-128 \dots 127]$, which causes problems in the following example,

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

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

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

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

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

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

The overflow error in Listing 5.20 can be understood in terms of the binary representation of integers: In binary, $130 = 10000010_2$, and this binary pattern is interpreted differently as `byte` and `sbyte`,

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

```
> 0b100000010uy;;  
val it : byte = 130uy  
> 0b100000010y;;  
val it : sbyte = -126y
```

That is, for signed bytes, the left-most bit is used to represent the sign, and since the addition of $100 = 01100100_2$ and $30 = 00011110_2$ 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.,

· remainder

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

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

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

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

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

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

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

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

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

· run-time error

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

a	b	a ~~~ b
false	false	false
false	true	true
true	false	true
true	true	false

Table 5.5: Boolean exclusive or truth table.

Listing 5.26, integerPown.fsx:
Integer exponent function.

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

which is equal to 2^5 .

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

· xor
· exclusive or

5.5 Floating point arithmetic

The set of reals is infinitely large, and since all computers have limited resources, it is not possible to represent it in their entirety. The various floating point types listed in Table 5.1 are finite subset reduced by sampling the space of reals. An in-depth description of floating point implementations can be found in Appendix A. The type `float` is the most common type.

Table ??, ??, and ?? gives examples operators and functions pre-defined for floating point types. For most addition, subtraction, multiplication, divisions, and negation the result straight forward.

The remainder operator for floats calculates the remainder after division and discarding the fractional part,

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

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

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

Listing 5.28, floatDivisionRemainderLossless.fsx:

Floating point division, truncation, and remainder is a lossless representation of a number.

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

<

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

Listing 5.29, floatDivisionByZero.fsx:

Floating point numbers include infinity and Not-a-Number.

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

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

Listing 5.30, floatImprecision.fsx:

Floating point arithmetic has finite precision.

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

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

Advice

5.6 Char and string arithmetic

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

· ASCIIbetical order

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

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

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

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

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

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

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

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

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

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

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

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

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`. There 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 `leftOp < rightOp` is as follows: we start by examining the first character, and if `leftOp.[0]` and `rightOp.[0]` are different, then the `leftOp < rightOp` is equal to `leftOp.[0] < rightOp.[0]`. E.g., "`milk`" `<` "`abs`" is the same as `'m' < 'a'` is false, since the letter 'm' does not come before the letter 'a' in the alphabet, or more precisely, the codepoint of 'm' is not less than the codepoint of 'a'. If `leftOp.[0]` and `rightOp.[0]` are equal, then we move onto the next letter and repeat the investigation, e.g., "`abe`" `<` "`abs`" is true, since "`ab`" = "`ab`" is true and `'e' < 's'` is true. If we reach the end of either of the two strings, then the short is smaller than the larger, e.g., "`abs`" `<` "`absalon`" is true, while "`abs`" `<` "`abs`" is false. The `<=`, `>`, and `>=` operators are defined similarly.

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

. [], 32
ReadKey, 98
ReadLine, 98
Read, 98
System.Console.ReadKey, 98
System.Console.ReadLine, 98
System.Console.Read, 98
System.Console.WriteLine, 98
System.Console.Write, 98
WriteLine, 98
Write, 98
abs, 23
acos, 23
asin, 23
atan2, 23
atan, 23
bignum, 20
byte[], 20
byte, 20
ceil, 23
char, 17
cosh, 23
cos, 23
decimal, 20
double, 20
eprintfn, 47
eprintf, 47
exn, 17
exp, 23
failwithf, 47
float32, 20
float, 17
floor, 23
fprintfn, 47
fprintf, 47
ignore, 47
int16, 20
int32, 20
int64, 20
int8, 20
int, 17
it, 17
log10, 23
log, 23
max, 23

min, 23
nativeint, 20
obj, 17
pown, 23
printfn, 47
printf, 45, 47
round, 23
sbyte, 20
sign, 23
single, 20
sinh, 23
sin, 23
sprintf, 47
sqrt, 23
stderr, 47, 98
stdin, 98
stdout, 47, 98
string, 17
tanh, 23
tan, 23
uint16, 20
uint32, 20
uint64, 20
uint8, 20
unativeint, 20
unit, 17

American Standard Code for Information Inter-
change, 135

and, 28
anonymous function, 42
array sequence expressions, 117
Array.toArray, 73
Array.toList, 73
ASCII, 135
ASCIIbetical order, 32, 135

base, 17, 131
Basic Latin block, 136
Basic Multilingual plane, 136
basic types, 17
binary, 131
binary number, 18
binary operator, 23
binary64, 131
binding, 13

- bit, 18, 131
- black-box testing, 77
- block, 38
- blocks, 136
- boolean and, 27
- boolean or, 27
- branches, 62
- branching coverage, 78
- bug, 76
- byte, 131

- character, 19
- class, 21, 33
- code point, 19, 136
- compiled, 10
- computation expressions, 68, 71
- conditions, 62
- Cons, 69
- console, 10
- coverage, 78
- currying, 43

- debugging, 12, 77, 83
- decimal number, 17, 131
- decimal point, 17, 131
- Declarative programming, 6
- digit, 17, 131
- dot notation, 23
- double, 131
- downcasting, 22

- EBNF, 17, 139
- efficiency, 76
- encapsulate code, 40
- encapsulation, 43, 49
- environment, 83
- exception, 30
- exclusive or, 30
- executable file, 10
- expression, 13, 22
- expressions, 6
- Extended Backus-Naur Form, 17, 139
- Extensible Markup Language, 52

- file, 97
- floating point number, 17
- format string, 13
- fractional part, 17, 22
- function, 15
- function coverage, 78
- Functional programming, 6, 108
- functional programming, 6
- functionality, 76
- functions, 6

- generic function, 41

- hand tracing, 83
- Head, 69
- hexadecimal, 131
- hexadecimal number, 18
- HTML, 55
- Hyper Text Markup Language, 55

- IEEE 754 double precision floating-point format, 131
- Imperativ programming, 108
- Imperative programming, 6
- implementation file, 10
- infix notation, 23
- infix operator, 22
- integer division, 29
- integer number, 17
- interactive, 10
- IsEmpty, 69
- Item, 69

- jagged arrays, 73

- keyword, 13

- Latin-1 Supplement block, 136
- Latin1, 135
- least significant bit, 131
- Length, 69
- length, 66
- lexeme, 15
- lexical scope, 15, 41
- lexically, 36
- lightweight syntax, 34, 37
- list, 68
- list sequence expression, 117
- List.Empty, 69
- List.toArray, 69
- List.toList, 69
- literal, 17
- literal type, 20

- machine code, 108
- maintainability, 77
- member, 21, 66
- method, 33
- mockup code, 83
- module elements, 128
- modules, 10
- most significant bit, 131
- Mutable data, 47

- namespace, 21
- namespace pollution, 122
- NaN, 133

- nested scope, 38
- newline, 20
- not, 28
- not a number, 133
- obfuscation, 68
- object, 33
- Object oriented programming, 108
- Object-oriented programming, 7
- objects, 7
- octal, 131
- octal number, 18
- operand, 40
- operands, 23
- operator, 23, 40
- or, 28
- overflow, 29
- overshadows, 39
- pattern matching, 119, 124
- portability, 77
- precedence, 23
- prefix operator, 23
- Procedural programming, 108
- procedure, 43
- production rules, 139
- ragged multidimensional list, 71
- raise an exception, 91
- range expression, 68
- reals, 131
- recursive function, 63
- reference cells, 50
- reliability, 76
- remainder, 29
- rounding, 22
- run-time error, 30
- scientific notation, 18
- scope, 38
- script file, 10
- script-fragment, 15
- script-fragments, 10
- Seq.initInfinite, 117
- Seq.item, 115
- Seq.take, 115
- Seq.toArray, 117
- Seq.toList, 117
- side-effect, 72
- side-effects, 43, 50
- signature file, 10
- slicing, 72
- software testing, 77
- state, 6
- statement, 13
- statement coverage, 78
- statements, 6, 108
- states, 108
- stopping criterium, 64
- stream, 97
- string, 13, 19
- Structured programming, 7
- subnormals, 133
- Tail, 69
- tail-recursive, 64
- terminal symbols, 139
- tracing, 83
- truth table, 28
- tuple, 66
- type, 13, 17
- type casting, 21
- type declaration, 13
- type inference, 12, 13
- type safety, 40
- unary operator, 23
- underflow, 29
- Unicode, 19
- unicode general category, 136
- Unicode Standard, 136
- unit of measure, 121
- unit testing, 77
- unit-less, 121
- unit-testing, 12
- upcasting, 22
- usability, 76
- UTF-16, 136
- UTF-8, 136
- variable, 47
- verbatim, 21
- white-box testing, 77, 78
- whitespace, 20
- whole part, 17, 22
- wild card, 36
- word, 131
- XML, 52
- xor, 30
- yield bang, 115