Learning to program with F#

Jon Sporring

July 24, 2016

Contents

Chapter 5

Constants, tuples, and types

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.

· basic types

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

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

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

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

Table 5.1: List of basic types. The most commonly used types are highlighted in bold. For at description of integer see Appendix A.1, for floating point numbers see Appendix A.2, for ASCII and Unicode characters see Appendix B, for objects see Chapter 18, and for exceptions see Chapter 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 token `e` may be an `dInt` or a `dFloat`, but the exponent value must be an `dInt`.

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

· bit
· binary number
· octal number
· hexadecimal number

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

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

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

· character
· Unicode
· code point

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

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

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

· string

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

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

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

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

Examples are `"a"`, `"this is a string"`, and `"-&#\@"`. Newlines and following white spaces are taken literally, but may be ignored by a preceding \character. Further examples of strings are,

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

Listing 5.3: fsharpi, Examples of string literals.

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

· literal type

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

Listing 5.4: fsharpi, Named and implied literals.

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

Table 5.3: List of literal type. No spacing is allowed between the literal and the prefix or suffix. `[]` notation is for lists, see Chapter 10. *`bignum` does not yet have an implementation for `dInt ("Q"|"R"|"Z"|"N"|"G")` in Mono.

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

· verbatim

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

Listing 5.5: fsharpi, Examples of a string literal.

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

Advice!

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

· type casting

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

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

which is a `float`, since when `float` is given an argument, then it acts as a function rather than a type, and for the integer `3` it returns the floating point number `3.0`. For more on functions see Chapter 7. Boolean values are often treated as the integer values 0 and 1, but no short-hand function names exists for their conversions. Instead use,

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

Listing 5.7: fsharpi, Casting booleans.

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

· member
· class
· namespace

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

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

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

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

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

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

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

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

5.2 Operators on basic types

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

· expression
· infix operator

```
bool = "true" | "false"
const = byte | sbyte | uint8 | int8 | int16 | uint16 | int | int32 | uint32 |
    int64 | uint64 | bignum | navitei nt | unativeint | float | double | single
     | float32 | decimal | char | string | byte [] | bool | "()"
expr =
  const (* constant value *)
  | "(" expr ")" (* block expression *)
  | expr operator expr (* infix operation *)
  | operator expr (* prefix operation *)
  | expr expr (* function application *)
  | ...
```

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.

· 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

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

Operator	`op1`	`op2`	Expression	Result	Description
`op1 + op2`	ints	ints	`5 + 2`	`7`	Addition
	floats	floats	`5.0 + 2.0`	`7.0`	
	chars	chars	`'a' + 'b'`	`'\195'`	Addition of codes
	strings	strings	`"ab" + "cd"`	`"abcd"`	Concatenation
`op1 - op2`	ints	ints	`5 - 2`	`3`	Subtraction
	floats	floats	`5.0 - 2.0`	`3.0`	
`op1 * op2`	ints	ints	`5 * 2`	`10`	Multiplication
	floats	floats	`5.0 * 2.0`	`10.0`	
`op1 / op2`	ints	ints	`5 / 2`	`2`	Integer division
	floats	floats	`5.0 / 2.0`	`2.5`	Division
`op1 % op2`	ints	ints	`5 % 2`	`1`	Remainder
	floats	floats	`5.0 % 2.0`	`1.0`	
`op1 ** op2`	floats	floats	`5.0 ** 2.0`	`25.0`	Exponentiation
`op1 && op2`	bool	bool	`true && false`	`false`	boolean and
`op1 || op2`	bool	bool	`true || false`	`false`	boolean or
`op1 &&& op2`	ints	ints	`0b1010 &&& 0b1100`	`0b1000`	bitwise bool and
`op1 ||| op2`	ints	ints	`0b1010 ||| 0b1100`	`0b1110`	bitwise boolean or
`op1 ^^^ op2`	ints	ints	`0b1010 ^^^ 0b1101`	`0b0111`	bitwise boolean exclusive or
`op1 <<< op2`	ints	ints	`0b00001100uy <<< 2`	`0b00110000uy`	bitwise shift left
`op1 >>> op2`	ints	ints	`0b00001100uy >>> 2`	`0b00000011uy`	bitwise and
`+op1`	ints		`+3`	`3`	identity
	floats		`+3.0`	`3.0`	
`-op1`	ints		`-3`	`-3`	negation
	floats		`-3.0`	`-3.0`	
`not op1`	bool		`not true`	`false`	boolean negation
`~~~op1`	ints		`~~~0b00001100uy`	`0b11110011uy`	bitwise boolean negation

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

Operator	`op1`	`op2`	Expression	Result	Description

`op1 < op2`	bool
	ints
	floats
	chars
	strings
		bool
		ints
		floats
		chars
		strings
			`true < false`
			`5 < 2`
			`5.0 < 2.0`
			`'a' < 'b'`
			`"ab" < "cd"`
				`false`
				`false`
				`false`
				`true`
				`true`
					Less than

`op1 > op2`	bool
	ints
	floats
	chars
	strings
		bool
		ints
		floats
		chars
		strings
			`true > false`
			`5 > 2`
			`5.0 > 2.0`
			`'a' > 'b'`
			`"ab" > "cd"`
				`true`
				`true`
				`true`
				`false`
				`false`
					Greater than

`op1 = op2`	bool
	ints
	floats
	chars
	strings
		bool
		ints
		floats
		chars
		strings
			`true = false`
			`5 = 2`
			`5.0 = 2.0`
			`'a' = 'b'`
			`"ab" = "cd"`
				`false`
				`false`
				`false`
				`false`
				`false`
					Equal

`op1 <= op2`	bool
	ints
	floats
	chars
	strings
		bool
		ints
		floats
		chars
		strings
			`true <= false`
			`5 <= 2`
			`5.0 <= 2.0`
			`'a' <= 'b'`
			`"ab" <= "cd"`
				`false`
				`false`
				`false`
				`true`
				`true`
					Less than or equal

`op1 >= op2`	bool
	ints
	floats
	chars
	strings
		bool
		ints
		floats
		chars
		strings
			`true >= false`
			`5 >= 2`
			`5.0 >= 2.0`
			`'a' >= 'b'`
			`"ab" >= "cd"`
				`true`
				`true`
				`true`
				`false`
				`false`
					Greater than or equal

`op1 <> op2`	bool
	ints
	floats
	chars
	strings
		bool
		ints
		floats
		chars
		strings
			`true <> false`
			`5 <> 2`
			`5.0 <> 2.0`
			`'a' <> 'b'`
			`"ab" <> "cd"`
				`true`
				`true`
				`true`
				`true`
				`true`
					Not Equal

Table 5.5: Comparison operators on basic types. Ints, floats, chars, and strings means all built-in integer types etc..

Type	Function name	Example	Result	Description
Ints and floats	`abs`	`abs -3`	`3`	Absolute value
Floats	`acos`	`acos 0.8`	`0.6435011088`	Inverse cosine
Floats	`asin`	`asin 0.8`	`0.927295218`	Inverse sinus
Floats	`atan`	`atan 0.8`	`0.6747409422`	Inverse tangent
Floats	`atan2`	`atan2 0.8 2.3`	`0.3347368373`	Inverse tangentvariant
Floats	`ceil`	`ceil 0.8`	`1.0`	Ceiling
Floats	`cos`	`cos 0.8`	`0.6967067093`	Cosine
Floats	`cosh`	`cosh 0.8`	`1.337434946`	Hyperbolic cosine
Floats	`exp`	`exp 0.8`	`2.225540928`	Natural exponent
Floats	`floor`	`floor 0.8`	`0.0`	Floor
Floats	`log`	`log 0.8`	`-0.2231435513`	Natural logarithm
Floats	`log10`	`log10 0.8`	`-0.09691001301`	Base-10 logarithm
Ints, floats, chars, and strings	`max`	`max 3.0 4.0`	`4.0`	Maximum
Ints, floats, chars, and strings	`min`	`min 3.0 4.0`	`3.0`	Minimum
Ints	`pown`	`pown 3 2`	`9`	Integer exponent
Floats	`round`	`round 0.8`	`1.0`	Rounding
Ints and floats	`sign`	`sign -3`	`-1`	Sign
Floats	`sin`	`sin 0.8`	`0.7173560909`	Sinus
Floats	`sinh`	`sinh 0.8`	`0.8881059822`	Hyperbolic sinus
Floats	`sqrt`	`sqrt 0.8`	`0.894427191`	Square root
Floats	`tan`	`tan 0.8`	`1.029638557`	Tangent
Floats	`tanh`	`tanh 0.8`	`0.6640367703`	Hyperbolic tangent

Table 5.6: Predefined functions for arithmetic operations

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

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

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 5.10: fsharpi, A simple arithmetic expression.

Here, the addition and multiplication functions are shown in *infix notation* with the *operator* tokens `+` and `*`. To arrive at the resulting value 23, F# has to decide in which order to perform the calculation. 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[4]

· infix notation
· operator
· precedence

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

Listing 5.11: fsharpi, Precedences rules define implicite parentheses.

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

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

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

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

5.3 Booleans

Boolean arithmetic is the basis of almost all computers and particularly important for controlling program flow, which will be discussed in Chapter 9. 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 5.12: fsharpi, Boolean operators and truth tables.

Spacing produced using the `printfn` function is not elegant. In Section **??** we will discuss better options for producing more beautiful output. Notice, that the arguments for `printfn` was given on the next line. Here it is important to use *indentation* to indicate continuation of the line. See Section **??** for more.

· indentation

5.4 Integers and Reals

The set of integers and reals are infinitely large, and since all computers have limited resources, it is not possible to represent these sets in their entirety. The various integer and floating point types listed in Table 5.1 are finite subset where the integer types have been reduced by limiting their ranges and the floating point types have been reduced by sampling the space of reals. An in-depth description of integer and floating point implementations can be found in Appendix A. The `int` and `float` are the most common types.
Table 5.4 gives examples of how to use the operators for integers and floats, and most work like a standard calculator would.
For integers the following arithmetic operators are defined:

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

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

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

Listing 5.13: fsharpi, binary integer division and remainder operators.

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

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

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

Listing 5.14: fsharpi, binary division and remainder is a lossless representation of an integer.

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

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

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

Listing 5.15: fsharpi, integer exponentiation function, and the irrelevant identifier.

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

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

val it : int = 32
```

Listing 5.16: fsharpi, the equivalent to the irrelevant identifier.

Rumor has it, that the identifier `it` is an abbreviation for 'irrelevant'.
Performing arithmetic operations on `int` types requires extra care, since the result may cause *overflow*, *underflow*, and even exceptions, e.g., the range of the integer type `sbyte` is $[-128\ldots127]$, which causes problems in the following example, · overflow · underflow

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

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

Listing 5.17: fsharpi, adding integers may cause overflow.

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

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

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

Listing 5.18: fsharpi, subtracting integers may cause underflow

Notice that neither overflow nor underflow error gave rise to an error message, which is why such bugs are difficult to find. Dividing any non-zero number with 0 is infinite, which is also outside the domain of any of the integer types, but in this case, F# casts an*exception*, · exception

```
> 3/0;;
System.DivideByZeroException: Attempted to divide by zero.
  at <StartupCode$FSI_0007>.$FSI_0007.main@ () <0x6b78180 + 0x0000e> in <
      filename unknown>:0
  at (wrapper managed-to-native) System.Reflection.MonoMethod:InternalInvoke (
      System.Reflection.MonoMethod,object,object[],System.Exception&)
  at System.Reflection.MonoMethod.Invoke (System.Object obj, BindingFlags
      invokeAttr, System.Reflection.Binder binder, System.Object[] parameters,
       System.Globalization.CultureInfo culture) <0x1a55ba0 + 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
Integers can also be written in binary, octal, or hexadecimal format using the prefixes `0b`, `0o`, and `0x`, e.g.,

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

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

Listing 5.20: fsharpi, integer types may be specified as binary, octal, and hexadecimal numbers.

For a description of binary representations see Appendix A.1. The overflow error in Listing 5.17 can be understood in terms of the binary representation of integers: In binary, $130 = 10000010_2$, and this binary pattern is interpreted differently as `byte` and `sbyte`,

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

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

Listing 5.21: fsharpi, the left most bit is interpreted differently for signed and unsigned integers, which gives rise to potential overflow errors.

That is, for signed bytes, the left-most bit is used to represent the sign, and since the addition of $100 = 01100100_2$ and $30 = 00011110_b$ is $130 = 10000010_2$ causes the left-most bit to be used, then this is wrongly interpreted as a negative number, when stored in an `sbyte`.
For binary arithmetic on integers, the following operators are available:

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

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

`op1 &&& op2`: Bitwise 'and', returns the result of taking the boolean 'and' operator position-wise.

`op ||| op`: Bitwise 'or', as 'and' but using the boolean 'or' operator

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

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

position-wise.

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

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

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

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

we must consider the 8-bit binary form of the unsigned integers: $195 = 11000011_2$, $134 = 10000110_2$, $97 = 01100001_2$, and $60 = 00111100_2$, which agrees with the definitions. [5]
For floating point numbers the following arithmetic operators are defined:

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

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

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

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

Listing 5.23: fsharpi, unary floating point negation operator.

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

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

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

Listing 5.24: fsharpi, binary floating point addition, subtraction, multiplication, and division operators.

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

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

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

Listing 5.25: fsharpi, binary floating point division and remainder operators.

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

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

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

Listing 5.26: fsharpi, floating point division, truncation, and remainder is a lossless representation of a number.

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

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

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

val a : float = 32.0
```
Listing 5.27: fsharpi, binary floating point exponentiation.

which is equal to 2^5.

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

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

val a : float = infinity
val b : float = nan
```
Listing 5.28: fsharpi, floating point numbers include infinity and Not-a-Number

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

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

val a : float = 357.8
val b : float = 357.9
val c : float = 358.0
val d : float = 5.684341886e-14
```
Listing 5.29: fsharpi, floating point arithmetic has finite precision.

Hence, although `c` appears to be correctly calculated, by the subtraction we see, that the value bound in `c` is not exactly the same as `358.0`, and the reason is that the neither `357.8` nor `0.1` are exactly representable as a `float`, which is why the repeated addition accumulates a small representation error.

5.5 Chars and Strings

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

· ASCIIbetical order

```
> char (int 'z' - int 'a' + int 'A');;
val it : char = 'Z'
```
Listing 5.30: fsharpi, converting case by casting and integer arithmetic.

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

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

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

Listing 5.31: fsharpi, example of string concatenation.

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

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

val a : string = "abcdefg"
val b : char = 'a'
val c : char = 'd'
```

Listing 5.32: fsharpi, example of string indexing.

The *dot notation* is an example of Structured programming, where technically `a` is an immutable *object* of *class* `string`, and `[]` is an object *method*. For more on object, classes, and methods see Chapter 18. 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,

· dot notation
· object
· class
· method

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

val a : string = "abcdefg"
val l : int = 7
val first : char = 'a'
val last : char = 'g'
```

Listing 5.33: fsharpi, string length attribute and string indexing.

Notice, since index counting starts at 0, and the string length is 7, then the index of the last character is 6. An alternative notation for indexing is to use the property `Char`, and in the example `a.[3]` is the same as `a.Char 3`. The is a long list of built-in functions in `System.String` for working with strings, some of which will be discussed in Chapter 15.1.

5.6 Unit of Measure

F# allows for assigning *unit of measure* to the following types,

· unit of measure

`sbyte`, `int`, `int16`, `int32`, `int64`, `single`, `float32`, `float`, and `decimal`.

by using the syntax,

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

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

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

[<Measure>]
type m
```

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

Listing 5.34: fsharpi, floating point and integer numbers may be assigned unit of measures.

However, if we mixup unit of measures under addition, then we get an error,

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

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

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

Listing 5.35: fsharpi, unit of measures adds an extra layer of types for syntax checking at compile time.

Unit of measures allow for `*`, `/`, and `^`[6] for multiplication, division and exponentiation. Values with units can be casted to *unit-less* values by casting, and back again by multiplication as,

· unit-less

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

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

Listing 5.36: fsharpi, type casting unit of measures.

Compound symbols can be declared as,

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

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

Listing 5.37: fsharpi, aggregated unit of measures.

For fans of the metric system there is the International System of Units, and these are built-in in `Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols` and give in Table 5.9. Hence, using the predefined unit of seconds, we may write,

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

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

Table 5.9: International System of Units.

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

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

Listing 5.38: fsharpi, SI unit of measures are built-in.

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

· open

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

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

Listing 5.39: fsharpi, simpler syntax by importing, but beware of namespace pollution.

The `open` keyword should be used with care, since now all the bindings in `Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols` have been imported into the present scope, and since we most likely do not know, which bindings have been used by the programmers of `Microsoft.FSharp.Data.UnitSystems.SI.UnitSymbols`, we do not know which identifiers to avoid, when using `let` statements. We have obtained, what is known as *namespace pollution*. Read more about namespaces in Part IV. Using unit of measures is advisable for calculations involving real-world values, since some semantical errors of arithmetic expressions may be discovered by checking the resulting unit of measure.
[7]

· namespace pollution

[7]**add comparison operators!**

Chapter 6

Identifiers, functions, and variables

An identifier is bound to an expression by the syntax,

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

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

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

Thus, examples of identifiers are `a`, `theCharacter9`, `Next_Word`, `_tok`. The for the full definition, `letter = Lu | Ll | Lt | Lm | Lo | Nl` and `special-char = Pc | Mn | Mc | Cf`, which referes to the Unicode general categories described in Appendix B.3, and there are currently 19.345 possible Unicode code points in the `letter` category and 2.245 possible Unicode code points in the `special-char` category. An identifier must not be a keyword or a reserved-keyword, shown in Figure 6.1 and 6.2. The binding may be mutable, which will be discussed in Section 6.2, and the binding may only be for the last expression as indicated by the *in* keyword. The simplest example of an expression is a *literal*, i.e., a constant such as the number 3. · in · literal
A less common notation is to define bindings for expressions using the *in* keyword, e.g., · in

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

```
9.0
```

Listing 6.1: numbersIn.fsx - The identifier `p` is only bound in the nested scope following the keyword `in`.

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

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

Figure 6.1: List of keywords in F#.

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

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

6.1 Values (Constant bindings)

When specifying the type, the type and the literal form must match, i.e., mixing types and literals gives an error,

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

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

/Users/sporring/repositories/fsharpNotes/stdin(50,17): error FS0001: This
    expression was expected to have type
    float
but here has type
    int
```

Listing 6.2: fsharpi, binding error due to type mismatch.

since the left-hand-side is an identifier of type float, while the right-hand-side is a literal of type integer.

6.2 Variables (Mutable bindings)

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

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

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

· Mutable data
· variable

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

```
5
-3
```

Listing 6.3: mutableAssignReassingShort.fsx - A variable is defined and later reassigned a new value.

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

· `<-`

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

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

Listing 6.4: fsharpi, example of changing the content of a variable.

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

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

```



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

Listing 6.5: mutableAssignReassingTypeError.fsx - Assignmetn type mismatching causes a compile time error.

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

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

```
5
6
```

Listing 6.6: mutableAssignIncrement.fsx - Variable increment is a common use of variables.

which is an example we will return to many times later in this text.
1

2

. . .

[1]**Somewhere I should talk about whitespaces and newlines Spec-4.0 Chapter 3.1**
[2]**Somewhere I should possibly talk about Lightweight Syntax, Spec-4.0 Chapter 15.1**

Chapter 7

Functions and procedures (function bindings)

Function definition follows the same syntax as literal binding,

```
"let" ["rec"] ident valIdent {valIdent} [":" type] "=" expr ["in" expr]
valident = ident | "(" ident ":" type ")"
```

or specify the type of the function at point of definition using the notation,

```
  "let" name argWType { argWType } [ ":" type ] "=" expr
  argWType = arg | "(" arg ":" type ")"
```

where not all types need to be declared, just sufficent for F# to be able to infer the types for the full statement. In the example, one sufficent specification is,

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

val sum : x:float -> y:float -> float

> let c = sum 357.6 863.4;;

val c : float = 1221.0
```

Listing 7.1: fsharpi

but alternatively we could have specified the type of the result,

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

or even just one of the arguments,

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

In both cases, since the + *operator* is only defined for *operands* of the same type, then when the type of either the result, any or both operands are declared, then the type of the remaining follows directly. A function that elegantly implements the incrementation operation may be constructed as,

· operator
· operand

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

```
1
2
3
```

Listing 7.2: mutableAssignIncrementEncapsulation.fsx -

[1] Here the output of `incr` is an anonymous function, that takes no argument, increments the variable of `incr` and returns the new value of the counter. This construction is called *encapsulation*, since the variable `counter` is hidden by the function `incr` from the user, i.e., the user need not be concerned with how the increment operator is implemented and the variable name used by `incr` does not clutter the scope where it is used.

· encapsulation

Variables cannot be returned from functions, that is,

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

```
0
```

Listing 7.3: mutableAssignReturnValue.fsx -

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

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

```


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

Listing 7.4: mutableAssignReturnVariable.fsx -

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

· reference cells

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

```
0
3
```

Listing 7.5: mutableAssignReturnRefCell.fsx -

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

· side-effects

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

```
let updateFactor factor =
  factor := 2

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

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

```
6
```

Listing 7.6: mutableAssignReturnSideEffect.fsx -

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

```
let updateFactor () =
  2

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

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

```
6
```

Listing 7.7: mutableAssignReturnWithoutSideEffect.fsx -

Here there can be no doubt in `multiplyWithFactor` that the value of 'a' is changing. Side-effects do have their use, but should in general be avoided at almost all costs, and in general it is advised to refrain from using ref cells.
A function is a mapping between an input and output domain. F# is a functional first programming language, and offers a number of alternative methods for specifying parameters. A simple example is,

```
let mul (x, y) = x*y
let z = mul (3, 5)
printfn "%d" z
```

```
15
```

Listing 7.8: functionDeclarationMul.fsx -

which declares a function of a tuple and returns their multiplication. The types are inferred from its first use in the second line, i.e., `mul` is `val mul :  x:int * y:int -> int`. An argument may be of generic type for input, which need not be inferred without sacrificing type safety, e.g.,

```
let second (x, y) = y
let a = second (3, 5)
printfn "%A" a
let b = second ("horse", 5.0)
printfn "%A" b
```

```
5
5.0
```

Listing 7.9: functionDeclarationGeneric.fsx -

Here the function `second` does not use the first element in the tuple, `x`, and the type of the second element, `y`, can safely be anything.
Functions may be anonymously declared using the `fun` keyword,

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

```
5
```

Listing 7.10: functionDeclarationAnonymous.fsx -

Anonymous functions are often used as arguments to other functions, e.g.,

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

```
18
```

Listing 7.11: functionDeclarationAnonymousAdvanced.fsx -

This is a powerfull concept, but can make programs hard to read, and overly use is not recommended.
Functions may be declared using pattern matching, which is a flexible method for declaring output depending on conditions on the input value. The most common pattern matching method is by use of the `match with` syntax,

```
let rec factorial n =
  match n with
  | 0 -> 1
  | 1 -> 1
  | _ -> n * (factorial (n - 1))

printfn "%d" (factorial 5)
```

```
120
```

Listing 7.12: functionDeclarationMatchWith.fsx -

A short-hand only for functions of 1 parameter is the `function` syntax,

```
let rec factorial = function
  | 0 -> 1
  | 1 -> 1
  | n -> n * (factorial (n - 1))

printfn "%d" (factorial 5)
```

```
120
```

Listing 7.13: functionDeclarationFunction.fsx -

Note that the name given in the match, here `n`, is not used in the first line, and is arbitrary at the line of pattern matchin, and may even be different on each line. For these reasons is this syntax discouraged.
Functions may be declared from other functions

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

```
15
6
```

Listing 7.14: functionDeclarationTupleCurrying.fsx -

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

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

```
15
6
```

Listing 7.15: functionDeclarationCurrying.fsx -

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

7.1 Procedures

A *procedure* is a generalisation of the concept of functions, and in contrast to functions procedures need not return values. An example, we've already seen is the `printfn`, which is used to print text on the console, but does not return a value. Coincidentally, since the console is a state, printing to it is a side-effect. Above we examined · procedure

```
let updateFactor factor =
  factor := 2
```

which also does not have a return value. Procedural thinking is useful for encapsulation, but is prone to side-effects and should be minimized by being replaced by functional thinking.
[2]

[2]**Maybe explain the printf function, Spec-4.0 Section 6.3.16 'printf' Formats, but also max and min comparison functions and math functions Section 18.2.2 and 18.2.4?**

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

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

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

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

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

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

'=' definition, e.g.,

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

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

',' concatenation, e.g.,

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

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

';' termination of line

'|' alternative options, e.g.,

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

'-' exception, e.g.,

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

here `consonant` are all letters except vowels.

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

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

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

Here the comments demonstrate, the relaxed modification.

Bibliography

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

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

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

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

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

Index