Learning to program with F#

Jon Sporring

September 15, 2016

Contents

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 the number 3, and if we type the number `3` in an interactive session at the input prompt, then F# responds as follows, · type · literal

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., · `int` · `it`

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, *`bool`* for boolean values, *`char`* for characters, and *`string`* for strings of characters are the most common types of literals. A table of all *basic types* predefined in F# is given in Table 5.1. Besides these built-in types, F# is designed such that it is easy to define new types. · `float` · `bool` · `char` · `string` · basic types

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.

Humans like to use the *decimal number* system for representing numbers. Decimal numbers are *base* 10, which that a value is represented as two sequences of decimal 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 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*. As an example `35.7` is a decimal number, whose value is $3 \cdot 10^1 + 5 \cdot 10^0 + 7 \cdot 10^{-1}$, and `128` is an integer, whose value is $1 \cdot 10^2 + 2 \cdot 10^1 + 8 \cdot 10^0$. 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#. In EBNF, the grammar describing a decimal number is,

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

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*)
```

In EBNF `dDigit`, `dInt`, and `dFloat` are names of tokens, while `"0"`, `"1"`, ..., `"9"`, and `"."` are terminals. Tokens and terminals together with formatting rules describe possible sequences, which are valid. E.g., a `dDigit` is defined by the = notation to be either `0` or `1` or ... or `9`, as signified by the `|` syntax. The definition of a token is ended by a `;`. The `"{ }"` in EBNF signfies zero or more repetitions of its content, such that a `dInt` is, e.g., `dDigit`, `dDigit dDigit`, `dDigit dDigit dDigit dDigit` and so on. Since a `dDigit` is any decimal digit, we conclude that `3`, `45`, and `0124972930485738` are examples of `dInt`. A `dFloat` is the concatenation of one or more digits, a dot, and zero or more digits, such as `0.4235`, `3.`, but not `.5` nor `.`. Sometimes EBNF implicitly allows for spaces between tokens and terminals, so here we have used the comments notation `(* *)` to explicitly remind ourselves, that no spaces are allowed between the whole part, decimal point, and the fractional part. A complete description of EBNF is given in Appendix 3.

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

· scientific notation

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. Internally, programs and data is all represented as bits, hence F# has a strong support for binary numbers. 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 where each octal digit can be represented by exactly 3 bits, while hexadecimal numbers uses 16 as basis, and where each hexadecimal digit can be written in binary using exactly 4 bits. The hexadecimal digits uses `0`–`9` to represent the values 0–9 and `a`–`f` in lower or alternatively upper case to represent the values 10-15. 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

Listing 5.5: Binary, hexadecimal, 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 the value 367 in base 10 may be written as a `dInt` integer as `367`, as a `bitInt` binary number as `0b101101111`, as a `octInt` octal number as `0o557`, and as a `hexInt` hexadecimal number as `0x16f`. 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

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.

specification for the first 256 code points, and the hexadecimal escape codes `\uXXXX`, `\UXXXXXXXX` allow for the full specification of any code point. Examples of a `char` are `'a'`, `'_'`, `'\n'`, and `'\065'`.

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

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"
```

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 |`	`"a \"quote\".\n"`
	`'@"' {(char − ('"' | '\"' )) | '""'} '"' |`	`@"a ""quote"".\n"`

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

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

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

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., · verbatim

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., · typecasting

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 part of Structured programming to be discussed in Part IV.

· member
· class
· namespace

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 the set of 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,

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

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. In the example, `+` is the operator, and it is an infix operator, since it takes values on its left and right side. The grammar for expressions are defined recursively, and some of it is given by,

· expression
· infix operator

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, e.g., in the definition of `expression`, the token `expression` occurs both on the left and the right side of the = symbol. 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 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 E.1 and E.2 and a range of mathematical functions shown in Table E.3.

· 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,

· precedence

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

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 E.5, 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 implicite 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
```

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 making a simple scripts.**

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. Basic operations on boolean values are '*and*', '*or*', and '*not*', which in F# is written as the binary operators `&&`, `||`, and the function `not`. Since the domain of boolean values is so small, then all possible combination of input on these values can be written on tabular form, known as a *truth table*, and the truth tables for the basic boolean operators

· and
· or
· not
· truth table

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.

and functions is shown in Table 5.4. A good mnemonics for remembering the result of the 'and' and 'or' operators is to use 1 for true, 0 for false, multiplication for the boolean 'and' operator, and addition for boolean 'or' operator, e.g., true and false in this mnemonic translates to $1 \cdot 0 = 0$, and the results translates back to the boolean value false. In F# the truth table for the basic boolean operators is reproduced by,

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 = ()
```

In Listing 5.19 we used the `printfn` function, to present the results of many expressions on something that resembles a tabular form. The spacing produced using the `printfn` function is not elegant, and 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. The indentation is an important part of telling F#, which part of what you write belongs together. This is an example of the so-called lightweight syntax. Generally, F# ignores newlines and whitespaces except when using the ligthweight syntax, and the examples of the difference between regular and lightweight syntax is 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 subsets reduced by limiting their ranges. An in-depth description of integer implementation can be found in Appendix A. The type `int` is the most common type.

Table E.1, E.2, and E.3 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 may cause *overflow* and *underflow*. E.g., the range of the integer type `sbyte` is $[-128\dots127]$, which causes problems in the following example,

· overflow
· underflow

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.

```
> 0b10000010uy;;
val it : byte = 130uy
> 0b10000010y;;
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 $30 = 00011110_b$ is $130 = 10000010_2$ causes the left-most bit to be used, then this is wrongly interpreted as a negative number, when stored in an `sbyte`. Similar arguments can be made explaining underflows.

The division and remainder operators, which discards the fractional part after division, and the *remainder* operator calculates the remainder after integer division, e.g.,

· integer division
· 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 that 7, and the difference is `7 % 3`.

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

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
false	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 while inserting 0's to right; `leftOp >>> rightOp`, which shifts the bit pattern of `leftOp` `rightOp` positions to the right while inserting 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. Floating point types are finite subsets 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 E.1, E.2, and E.3 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, floatImprecission.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, upcaseChar.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'
```

Since index counting starts at 0, and the string length is 7, then the index of the last character is 6. The is a long list of built-in functions in `System.String` for working with strings, some of which will be discussed in Chapter F.1.

The *dot notation* is an example of Structured programming, where technically speaking, the string `"abcdefg"` is an immutable *object* of *class* `string`, `[]` 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"\space = "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'`, which 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.

5.7 Programming intermezzo

Conversion of integers between decimal and binary form is a key concept in order to understand some of the basic properties of calculations on the computer. From binary to decimal is straight forward using the power-of-two algorithm, i.e., given a sequence of $n+1$ bits that represent an intager $b_n b_{n-1} \dots b_0$, where b_n and b_0 are the most and least significant bits, then the decimal value is calculated as,

$$v = \sum_{i=0}^{n} b_i 2^i \tag{5.1}$$

For example $10011_2 = 1 + 2 + 16 = 19$. From decimal to binary is a little more complex, but a simple divide-by-two algorithm exists. The key to understanding the divide-by-two algorithm is to realize that when you divide a number by two, then that is equivalent to shifting its binary representation 1 to the right. E.g., $10 = 1010_2$ and $10/2 = 5 = 110_2$. Odd numbers have $b_0 = 1$, e.g., $11_{10} = 1011_2$ and $11_{10}/2 = 5.5 = 101.1_2$. Hence, if we divide any number by two and get a non-integer number, then its least significant bit was 1. Another way to express this is that the least significant bit is the remainder after integer division by two. Sequential application of this idea leads directly to the divide-by-two algorithm. E.g., if we were to convert the number 11_{10} on decimal form to binary form we would perform the following steps:

$$\begin{tikzcd}
11 \text{ div } 2 = & 5, \arrow[dl] \arrow[dr] & 11 \text{ rem } 2 = & 1 \\
5 \text{ div } 2 = & 2, \arrow[dl] \arrow[dr] & 5 \text{ rem } 2 = & 1 \\
2 \text{ div } 2 = & 1, \arrow[dl] \arrow[dr] & 2 \text{ rem } 2 = & 0 \\
1 \text{ div } 2 = & 0, & 1 \text{ rem } 2 = & 1
\end{tikzcd}$$

Here we used div and rem to signify the integer division and remainder operators. The algorithms stops, when the result of integer division is zero. Reading off the remainder from below and up we find the sequence 1011_2, which is the binary form of the decimal number 11_{10}. Using interactive mode, we can calculate the same as,

Listing 5.35: Converting the number 11_{10} to binary form.

```
> printfn "(%d, %d)" (11 / 2) (11 % 2);;
(5, 1)
val it : unit = ()
> printfn "(%d, %d)" (5 / 2) (5 % 2);;
(2, 1)
val it : unit = ()
> printfn "(%d, %d)" (2 / 2) (2 % 2);;
(1, 0)
val it : unit = ()
> printfn "(%d, %d)" (1 / 2) (1 % 2);;
(0, 1)
val it : unit = ()
```

Thus, but reading the second integer-respons from `printfn` from below and up, we again obtain the binary form of 11_{10} to be 1011_2. For integers with a fractional part, the divide-by-two may be used on the whole part, while multiply may be used in a similar manner on the fractional part.

Chapter 6

Constants, functions, and variables

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

Problem 6.1:

For given set constants a, b, and c, solve for x in

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

To solve for x we use the quadratic formula from elementary algebra,

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

which gives the general solution for any values of the coefficients. Here, we will assume a positive discrimant, $b^2 - 4ac > 0$. In order to write a program, where the code may be reused later, we define a function `discriminant : float -> float -> float` , that is, a function that takes 3 arguments, `a`, `b`, and `c`, and calculates the distriminant. Details on function definition is given in Section 6.2. Likewise, we will define functions `positiveSolution : float -> float -> float` and `negativeSolution : float -> float -> float`, that also takes the polynomial's coefficients as arguments and calculates the solution corresponding to choosing the postive and negative sign for $\pm$ in the equation. Our solution thus looks like Listing 6.1.

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

```
let discriminant a b c = b ** 2.0 - 4.0 * a * c
let positiveSolution a b c = (-b + sqrt (discriminant a b c)) / (2.0 * a)
let negativeSolution a b c = (-b - sqrt (discriminant a b c)) / (2.0 * a)

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

```
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has discriminant 4.0 and solutions -1.0 and 1.0
```

Here, we have further defined names of values `a`, `b`, and `c` used as input to our functions, and the results of function application is bound to the names `d`, `xn`, and `xp`. The names of functions and values given here are examples of identifiers, and with these, we may reuse the quadratic formulas and calculated values later, while avoiding possible typing mistakes and reducing amount of code, which needs to be debugged.

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

· lightweight syntax

The use of identifiers is central in programming. For F# not to be confused by built-in functionality, identifiers must follow a specific grammar: An identifier must start with a letter, but can be followed by zero or more of letters, digits, and a range of special characters except SP, LF, and CR (space, line feed, and carriage return). An identifier must not be a keyword or a reserved-keyword listed in Figures 6.1. An identifier is a name for a constant, an expression, or a type, and it is defined by the following EBNF:

Keywords:

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.

Reserved keywords for possible future use:

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.

Symbolic keywords:

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

Reserved symbolic keywords for possible future:

~ and `.

Figure 6.1: List of (possibly future) keywords and symbolic keywords in F#.

Listing 6.2: to do

```
ident = (letter | "_") {letter | dDigit | specialChar};
longIdent = ident | ident "." longIdent; (*no space around "."*)

longIdentOrOp = [longIdent "."] identOrOp; (*no space around "."*)
identOrOp =
  ident
  | "(" infixOp | prefixOp ")"
  | "(*)";

dDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
letter = Lu | Ll | Lt | Lm | Lo | Nl; (*e.g. "A", "B" ... and "a", "b", ...*)
specialChar = Pc | Mn | Mc | Cf; (*e.g., "_"*)

codePoint = ?Any unicode codepoint?;
Lu = ?Upper case letters?;
Ll = ?Lower case letters?;
Lt = ?Digraphic letters, with first part uppercase?;
Lm = ?Modifier letters?;
Lo = ?Gender ordinal indicators?;
Nl = ?Letterlike numeric characters?;
Pc = ?Low lines?;
Mn = ?Nonspacing combining marks?;
Mc = ?Spacing combining marks?;
Cf = ?Soft Hyphens?;
```

Thus, examples of identifiers are `a`, `theCharacter9`, `Next_Word`, `_tok`. Typically, only letters from the english alphabet are used as `letter`, and only _ is used for `specialChar`, but the full definition referes to the Unicode general categories described in Appendix B.3, and there are currently 19.345 possible Unicode code points in the `letter` category and 2.245 possible Unicode code points in the `specialChar` category.

Expressions are a central concept in F#. An expression can be a mathematical expression, such as $3*5$, a function application, such as $f3$, and many other things. Central in this chapter is the binding of values and functions to indentifiers, which is done with the keyword `let`, using the following simplified syntax, e.g., `let a = 1.0`.

Expressions has an enormous variety in how they may be written, we will in this book gradually work through some of the more important facets. For this we will extend the EBNF notation with ellipses: `...`, to denote that what is shown is part of the complete EBNF production rule. E.g., the part of expressions, we will discuss in this chapter is specified in EBNF by,

Listing 6.3: to do

```
expr = ...
  | expr ":" type (*type annotation*)
  | expr ";" expr (*sequence of expressions*)
  | "let" valueDefn "in" expr (*binding a value or variable*)
  | "let" ["rec"] functionDefn "in" expr (*binding a function or operator*)
  | "fun" argumentPats "->" expr (*anonymous function*)
  | expr "<-" expr (*assignment*)

type = ...
  | longIdent (*named such as "int"*)

valueDefn = ["mutable"] pat "=" expr;

pat = ...
  | "_" (*wildcard*)
  | ident (*named*)
  | pat ":" type (*type constraint*)
  | "(" pat ")" (*paranthesized*)

functionDefn = identOrOp argumentPats [":" type] "=" expr;
argumentPats = pat | pat argumentPats;
```

In the following sections, we will work through this bit by bit.

6.1 Values

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

Listing 6.4: to do

```
expr = ...
  | "let" valueDefn "in" expr (*binding a value or variable*)
```

The let bindings defines relations between patterns pat and expressions expr for many different purposes. Most often the pattern is an identifier ident, which *let* defines to be an alias of the expression expr. The pattern may also be defined to have specific type using the *:* lexeme and a named type. The _ pattern is called the *wild card* pattern and, when it is in the value-binding, then the expression is evaluated but the result is discarded. The binding may be mutable as indicated by the keyword *mutable*, which will be discussed in Section 6.5, and the binding holds *lexically* for the last expression as indicated by the *in* keyword. For example, letting the identifier p be bound to the value 2.0 and using it in an expression is done as follows,

· let
· :
· wild card
· mutable
· lexically
· in

Listing 6.5, letValue.fsx:
The identifier p is used in the expression following the in keyword.

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

```
9.0
```

F# will ignore most newlines between lexemes, i.e., the above is equivalent to writing,

Listing 6.6, letValueLF.fsx:
Newlines after in make the program easier to read.

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

```
9.0
```

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

· lightweight syntax

Listing 6.7, letValueLightWeight.fsx:
Lightweight syntax does not require the in keyword, but expression must be aligned with the let keyword.

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

```
9.0
```

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

Listing 6.8, letValueLightWeightTypes.fsx:
Interactive mode also responds inferred types.

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

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

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

Listing 6.9, letValueTypeError.fsx:
Binding error due to type mismatch.

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

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

Here, the left-hand-side is defined to be an identifier of type float, while the right-hand-side is a literal of type integer.

An expression can be a sequence of expressions separated by the lexeme `;`, e.g.,

Listing 6.10, letValueSequence.fsx:
A value-binding for a sequence of expressions.

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

```
2.0
9.0
```

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

Listing 6.11, letValueSequenceLightWeight.fsx:
A value-binding for a sequence using lightweight syntax.

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

```
2.0
9.0
```

A key concept of programming is *scope*. In F#, the scope of a value-binding is lexically meaning that when F# determines the value bound to a name, it looks left and upward in the program text for the `let` statement defining it, e.g.,

· scope

Listing 6.12, letValueScopeLower.fsx:
Redefining identifiers is allowed in lower scopes.

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

```
 4
```

F# also has to option of using dynamic scope, where the value of a binding is defined by when it is used, and this will be discussed in Section 6.5.

Scopes are given levels, and scopes may be nested, where the nested scope has a level one lower than its parent.[1] F# distinguishes between the top and lower levels, and at the top level in the lightweight syntax, redefining values is not allowed, e.g.,

Listing 6.13, letValueScopeLowerError.fsx:
Redefining identifiers is not allowed in lightweight syntax at top level.

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

```



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

But using parentheses, we create a *block*, i.e., a *nested scope*, and then redefining is allowed, e.g.,

· block
· nested scope

[1]Todo: **Drawings would be good to describe scope**

Listing 6.14, letValueScopeBlockAlternative3.fsx:
A block may be created using parentheses.

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

```
4
```

In both cases we used indentation, which is good practice, but not required here. Bindings inside are not available outside the nested scope, e.g.,

Listing 6.15, letValueScopeNestedScope.fsx:
Bindings inside a scope are not available outside.

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

```



/Users/sporring/repositories/fsharpNotes/src/letValueScopeNestedScope.fsx
    (6,19): error FS0039: The value or constructor 'q' is not defined
```

Nesting is a natural part of structuring code, e.g., through function definitions to be discussed in Section 6.2 and flow control structures to be discussed in Chapter 8. Blocking code by nesting is a key concept for making robust code that is easy to use by others without the user necessarily needing to know the details of the inner workings of a block of code.

Defining blocks is useful for controlling the extend of a lexical scope of bindings. For example, adding a second `printfn` statement,

Listing 6.16, letValueScopeBlockProblem.fsx:
Overshadowing hides the first binding.

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

```
4
4
```

will print the value 4 last bound to the identifier `p`, since F# interprets the above as `let p = 3 in let p = 4 in (printfn "%A" p; printfn "%A" p)`. Had we intented to print the two different values of `p`, the we should have create a block as,

Listing 6.17, letValueScopeBlock.fsx:
Blocks allow for the return to the previous scope.

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

```
 4
 3
```

Here, the lexical scope of `let p = 4 in ...` is for the nested scope, which ends at `)`, returning to the lexical scope of `let p = 3 in ...`.

6.2 Non-recursive functions

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

· encapsulate code

Listing 6.18: to do

```
expr = ...
  | "let" functionDefn "in" expr (*binding a function or operator*)
```

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

Listing 6.19, letFunction.fsx:
An example of a binding of an identifier and a function.

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

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

and we see that the function is interpreted to have the type `val sum : x:float -> y:float -> float`. The `->` lexeme means a mapping between sets, in this case floats. The function is also a higher order function, to be discussed in detail below, and here it suffices to think of `sum` as a function that takes 2 floats as argument and returns a float.

Not all types need to be declared, just sufficient for F# to be able to infer the types for the full statement. In the example, one specification is sufficient, and we could just have specified the type of the result,

Listing 6.20: All types need most often not be specified.

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

or even just one of the arguments,

Listing 6.21: Just one type is often enough for F# to infer the rest.

```
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. As for values, lightweight syntax automatically inserts the keyword `in` and the lexeme `;`,

· operator

· operand

Listing 6.22, letFunctionLightWeight.fsx:
Lightweight syntax for function definitions.

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

```
1221.0
```

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

· type safety

Listing 6.23, functionDeclarationGeneric.fsx:
Typesafety implies that a function will work for any type, and hence it is generic.

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

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

Here, the function `second` does not use the first argument `x`, which therefore can be of any type, and which F# therefore calls `'a`, and the type of the second element, `y`, can also be of any type and not necessarily the same as `x`, so it is called `'b`. Finally the result is the same type as `y`, whatever it is. This is an example of a *generic function*, since it will work on any type.

· generic function

A function may contain a sequence of expressions, but must return a value. E.g., the quadratic formula may be written as,

Listing 6.24, identifiersExampleAdvance.fsx:
A function may contain sequences of expressions.

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

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

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

Here, we used the lightweight syntax, where the `=` identifies the start of a nested scope, and F# identifies the scope by indentation. The amount of space used for indentation is does not matter, but all lines following the first must use the same. The scope ends before the first line with the previous indentation or none. Notice how the last expression is not bound to an identifier, but is the result of the function, i.e., in contrast to many other languages, F# does not have an explicit keyword for returning values, but requires a final expression, which will be returned to the caller of the function. Note also that since the function `discriminant` is defined in the nested scope of `solution`, then `discriminant` cannot be called outside `solution`, since the scope ends before `let a = 1.0`.

Lexical scope and function definitions can be a cause of confusion as the following example shows,[2] · lexical scope

Listing 6.25, lexicalScopeNFunction.fsx:
Lexical scope means that $f(z) = 3x$ and not $4x$ at the time of calling.

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

```
6.0
```

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

[2]Todo: **Add a drawing or possibly a spell-out of lexical scope here.**

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

· anonymous function

Listing 6.26, functionDeclarationAnonymous.fsx:
Anonymous functions are functions as values.

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

```
5
```

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

Listing 6.27, functionDeclarationAnonymousAdvanced.fsx:
Anonymous functions are often used as arguments for other functions.

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

```
18
```

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

Advice

Functions may be declared from other functions

Listing 6.28, functionDeclarationCurrying.fsx:

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

```
15
6
```

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

· currying
Advice

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

· procedure

Listing 6.29, procedure.fsx:
A procedure is a function that has no return value, which in F# implies `()` as return value.

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

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

In F# this is automatically given the unit type as return value. Procedural thinking is useful for *encapsulation* of scripts, but is prone to *side-effects* and should be minimized by being replaced by functional thinking. More on side-effects in Section 6.5. **Procedural thinking is useful for encapsulation, but is prone to side-effects and should be minimized by being replaced by functional thinking.**

· encapsulation
· side-effects
Advice

6.3 User-defined operators

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

Listing 6.30, addOperatorNFunction.fsx:

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

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

All operator has this option, and you may redefine them and define your own operators, but in F# names of user-defined operators are limited by the following simplified EBNF:

Listing 6.31: Grammar for infix and prefix lexemes

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

The precedence rules and associativity of user-defined operators follows the rules for which they share prefixes with built-in rules, see Table E.6. E.g., `.*`, `+++`, and `<+` are valid operator names for infix operators, they have precedence as ordered, and their associativity are all left. Using `~` as the first character in the definition of an operator makes the operator unary and will not be part of the name. Examples of definitions and use of operators are,

Listing 6.32, operatorDefinitions.fsx:

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

```
13
16
true
2
```

Operators beginning with `*` must use a space in its definition, `( *` in order for it not to be confused with the beginning of a comment `(*`, see Chapter 7 for more on comments in code.

Beware, redefining existing operators lexically redefines all future uses of the operators for all types, hence **it is not a good idea to redefine operators, but better to define new.** In Chapter 20 we will discuss how to define type specific operators including prefix operators. Advice

6.4 The Printf function

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

Listing 6.33: to do

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

where a formatString is a string (simple or verbatim) with placeholders. The function printf prints formatString to the console, where all placeholder has been replaced by the value of the corresponding argument formatted as specified, e.g., in `printfn "1 2 %d" 3` the formatString is `"1 2 %d"`, and the placeholder is `%d`, and the `printf` replaced the placeholder with the value of the corresponding argument, and the result is printed to the console, in this case `1 2 3`. Possible formats for the placeholder are,

Listing 6.34: to do

```
placeholder = "%%" | ("%" [flags] [width] ["." precision] specifier) (* No
    spaces between rules *)
flags = ["0"] ["+"] [SP] (* No spaces between rules *)
width = ["-"] ("*" | [dInt]) (* No spaces between rules *)
specifier = "b" | "d" | "i" | "u" | "x" | "X" | "o" | "e" | "E" | "f" | "F" |
    "g" | "G" | "M" | "O" | "A" | "a" | "t"
```

There are specifiers for all the basic types and more as elaborated in Table 6.1. The placeholder can be given a specified with, either by setting a specific integer, or using the `*` character, indicating that the with is given as an argument prior to the replacement value. Default is for the value to be right justified in the field, but left justification can be specified by the `-` character. For number types, you can specify their format by: `"0"` for padding the number with zeros to the left, when righ justifying the number; `"+"` to explicitely show a plus sign for positive numbers; `SP` to enforce a space, where there otherwise would be a plus sign for positive numbers. For floating point numbers, the precision integer specifies the number of digits displayed of the fractional part. Examples of some of these combinations are,

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

Table 6.1: Printf placeholder string

Listing 6.35, printfExample.fsx:
Examples of printf and some of its formatting options.

```
let pi = 3.1415192
let hello = "hello"
printf "An integer: %d\n" (int pi)
printf "A float %f on decimal form, and on %e scientific form, and a char
    '%c'\n" pi pi
printf "A char '%c' and a string \"%s\"\n" hello.[0] hello
printf "Float using width 8 and 1 number after the decimal:\n"
printf "  \"%8.1f\" \"%8.1f\"\n" pi -pi
printf "  \"%08.1f\" \"%08.1f\"\n" pi -pi
printf "  \"% 8.1f\" \"% 8.1f\"\n" pi -pi
printf "  \"%-8.1f\" \"%-8.1f\"\n" pi -pi
printf "  \"%+8.1f\" \"%+8.1f\"\n" pi -pi
printf "  \"%8s\"\n\"%-8s\"\n" "hello" "hello"
```

```
An integer: 3
A char 'h' and a string "hello"
Float using width 8 and 1 number after the decimal:
  "     3.1" "    -3.1"
  "000003.1" "-00003.1"
  "     3.1" "    -3.1"
  "3.1     " "-3.1    "
  "    +3.1" "    -3.1"
  "   hello"
"hello   "
```

Function	Example	Description
`printf`	`printf "%d apples" 3`	Prints to the console, i.e., `stdout`
`printfn`		as `printf` and adds a newline.
`fprintf`	`fprintf stream "%d apples" 3`	Prints to a stream, e.g., `stderr` and `stdout` , which would be the same as `printf` and `eprintf`.
`fprintfn`		as `fprintf` but with added newline.
`eprintf`	`eprintf "%d apples" 3`	Print to `stderr`
`eprintfn`		as `eprintf` but with added newline.
`sprintf`	`printf "%d apples" 3`	Return printed string
`failwithf`	`failwithf "%d failed apples" 3`	prints to a string and used for raising an exception.

Table 6.2: The family of printf functions.

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

Listing 6.36, printfExampleAdvance.fsx:

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

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

The `%A` is special in that all built-in types including tuples, lists, and arrays to be discussed in Chapter 9 can be printed using this formatting string, but notice that the formatting performed includes the named literal string. The two formatting strings `%t` and `%a` are options for user-customizing the formatting, and will not be discussed further.

Beware, `formatString` is not a `string` but a `Printf.TextWriterFormat`, so to predefine a `formatString` as, e.g., `let str = "hello %s" in printf str "world"` will be a type error.

The family of printf is shown in Table 6.2. The function fprintf prints to a stream, e.g., stderr and stdout, of type System.IO.TextWriter. Streams will be discussed in further detail in Chapter 12. The function failwithf is used with exceptions, see Chapter 11 for more details. The function has a number of possible return value types, and for testing the *ignore* function ignores it all, e.g., ignore (failwithf "%d failed apples" 3) · ignore

6.5 Variables

The mutable in let bindings means that the identifier may be rebound to a new value using the <- lexeme, e.g.,[3] · <-

Listing 6.37: to do

```
expr = ...
  | expr "<-" expr (*assingment*)
```

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

Listing 6.38, mutableAssignReassingShort.fsx: A variable is defined and later reassigned a new value.

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

```
5
-3
```

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

Listing 6.39, mutableEqual.fsx: Common error - mistaking = and <- lexemes for mutable variables. The former is the test operator, while the latter is the assignment expression.

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

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

[3]Todo: **Discussion on heap and stack should be added here.**

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

Assignment type mismatches will result in an error,

**Listing 6.40, mutableAssignReassingTypeError.fsx:
Assignment type mismatching causes a compile time 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
```

I.e., once the type of an identifier has been declared or inferred, then it cannot be changed.

A typical variable is a counter of type integer, and a typical use of counters is to increment them, i.e., erasing a new value to be one more that its previous value. For example,

**Listing 6.41, mutableAssignIncrement.fsx:
Variable increment is a common use of variables.**

```
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
```

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

Listing 6.42, mutableAssignIncrementEncapsulation.fsx:

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

```
1
2
3
```

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

Variables implement dynamic scope, e.g., in comparison with the lexical scope, where the value of an identifier depends on which line in the program, an identifier is defined, dynamic scope depends on, when it is used. E.g., the script in Listing 6.25 defines a function using lexical scope and returns the number `6.0`, however, if `a` is made `mutable`, then the behaviour is different:

Listing 6.43, dynamicScopeNFunction.fsx:
Mutual variables implement dynamics scope rules. Compare with Listing 6.25.

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

```
8.0
```

Here, the respons is `8.0`, since the value of `a` changed befor the function `f` was called.

Variables cannot be returned from functions, that is,

[4]Todo: **Explain why this works!**

Listing 6.44, mutableAssignReturnValue.fsx:

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

```
0
```

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

Listing 6.45, mutableAssignReturnVariable.fsx:

```
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
```

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

Listing 6.46, mutableAssignReturnRefCell.fsx:

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

```
0
3
```

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,[5] · side-effects

[5]Todo: **Discuss side-effects!**

Listing 6.47, mutableAssignReturnSideEffect.fsx:

```
let updateFactor factor =
  factor := 2

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

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

```
6
```

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,

Listing 6.48, mutableAssignReturnWithoutSideEffect.fsx:

```
let updateFactor () =
  2

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

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

```
6
```

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.

6

[6]Todo: **Add something about mutable functions**

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