Chapter 3

Using F# as a Calculator

Abstract In the previous chapter, we introduced some key F# programming tools and

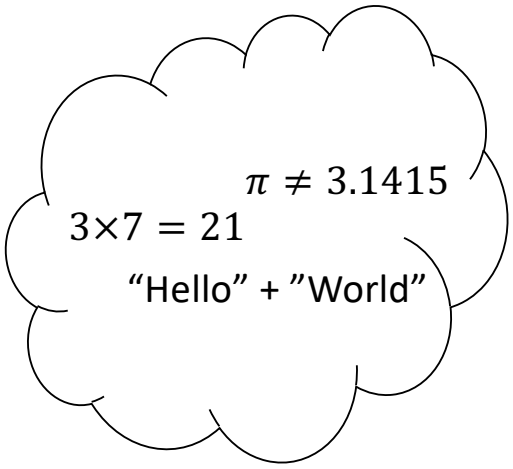


Fig. 3.1 The basis of F# are constants and expressions

concepts without going into depth on opportunities and limitations. In the following chapters, we will dive deeper into methods for solving problems by writing programs, and what facilities are available in F# to express these solutions. As a first step, we must acquaint ourselves with the basic building blocks of basic types, constants, and operators, and this chapter includes

- An introduction to the basic types and how to write constants of those types.
- Arithmetic of basic operations.

with these tools, you will be able to evaluate expressions as if F# were a simple calculator. Examples of problems, you will be able to solve after reading this chapter, is:

- What is the result of $3\cos(4 * \pi/180) + 4\sin(4 * \pi/180)$.
- Calculate how many characters are there in the text string “Hello World!”.
- What is the ASCII value of the character ’J’.
- How to convert between whole and binary numbers.

3.1 Literals and Basic Types

All programs rely on the processing of data, and an essential property of data is its *type*. A *literal* is a fixed value like the number 3, and if we type the number `3` in an interactive session at the input prompt, then F# responds as shown in Listing 3.1. What this means is that F# has inferred the type to be `int` and bound it to the

Listing 3.1: Typing the number 3.

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

identifier `it`. For more on binding and identifiers see Chapter 4. Types matter, since the operations that can be performed on integers, are quite different from those that can be performed on, e.g., strings. Therefore, the number 3 has many different representations as shown in Listing 3.2. Each literal represents the number 3, but

Listing 3.2: 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"
```

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 3.1. In addition to these built-in types, F# is designed such that it is easy to define new types.

Humans like to use the *decimal number* system for representing numbers. Decimal numbers are *base* 10, which means that a value is represented as two sequences of decimal digits separated by a *decimal point*, where each *digit* d has a position and a value $d \in \{0, 1, 2, \ldots, 9\}$. The part before the decimal point is called the *whole part* and the part after is called the *fractional part* of the number. An *integer* is a number with only a whole part and neither a decimal point nor a fractional part. 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*. Floating point numbers may alternatively be given

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 sbyte
	`uint8`	Synonymous with byte
	`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
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`	The value ()
Object	`obj`	An object
Exception	`exn`	An exception

Table 3.1 List of some of the basic types. The most commonly used types are underlined. For a description of integer see Appendix B.1, for floating point numbers see Appendix B.2, for ASCII and Unicode characters see Appendix C, for objects see Chapter 15, and for exceptions see Section 12.3.

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` $= 4 \cdot 10^2 = 400$.

The basic unit of information in almost all computers is the binary digit or *bit* for short. Internally, programs and data are all represented as bits, hence F# has 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 $101.01_2 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} = 5.25$. Subscripts are often used to indicate the base of a number, e.g., 101.01_2 and 101.01_{10} are different numbers. Since base 10 is so common, the subscript for base 10 numbers is often omitted.

Binary numbers are closely related to *octal* and *hexadecimal numbers*. Octals use 8 as their basis and hexadecimals use 16 as their basis. Each octal digit can be represented by exactly three bits, and each hexadecimal digit can be represented by exactly four bits. The hexadecimal digits use `0`–`9` to represent the values 0–9 and `a`–`f` in lower or alternatively upper case to represent the values 10-15. Thus, Octals and hexadecimals conveniently serve as shorthand for the much longer binary representation. As examples, the octal number 37_8 is $3 \cdot 8^1 + 7 \cdot 8^0 = 31$, and the hexadecimal number $f3_{16}$ is $15 \cdot 16^1 + 3 \cdot 16^0 = 243$.

To denote integers in bases different than 10, F# uses the prefix '0b' for binary, '0o' for octal, and '0x' for hexadecimal numbers. For example, the value 367_{10} may be written as an integer `367`, as a binary number `0b101101111`, as an octal number `0o557`, and as a hexadecimal number `0x16f`. In F#, the character sequences `0b12` and `ff` are not recognized as numbers.

A *character* is a *Unicode code point*, and character literals are enclosed in single quotation marks. Appendix C.3 contains more details on code points. The character type in F# is denoted *`char`*. Examples of characters are 'a', 'D', '3', and examples of non-characters are '23' and 'abc'. Some characters, such as the tabulation character, do not have a visual representation. These can still be represented as a character using *escape sequences*. A character escape sequence starts with "\" followed by letter for simple escapes such as `\t` for tabulation and `\n` for newline. Escape sequences can also be a numerical representation of a code point, and three versions exist: The trigraph `\DDD`, where `D` is a decimal digit, is used to specify the first 256 code points, the hexadecimal escape codes `\uXXXX`, where `X` is a hexadecimal digit, is used to specify the first 65536 code points, and `\UXXXXXXXX` is used to specify any of the approximately $4.3 \cdot 10^9$ possible code points. All escape sequences are shown in Table 3.2. Examples of `char` representations of the letter 'a' are: `'a'`, `'\097'`,

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 ('X' is any hexadecimal digit, and 'D' is any decimal digit)

Table 3.2 Escape characters. The escape code `\DDD` is sometimes called a tricode.

`'\u0061'`, `'\U00000061'`.

A *string* is a sequence of characters enclosed in double quotation marks. Examples are `"a"`, `"this is a string"`, and `"-&#\@"`. Note that the string `"a"` and the character `'a'` are not the same. Some strings are so common that they are given special names: One or more spaces `" "` is called *whitespace*, and both `"\n"` and `"\r\n"` are called *newline*. The escape-character "\" may be used to break a line in two. This and other examples are shown in Listing 3.3. Note that the response from `fsharpi` is shown in double quotation marks, but this is not part of the string.

F# supports *literal types*, where the type of a literal is indicated as a prefix or suffix as shown in Table 3.3.

Listing 3.3: 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	syntax	Examples	Value
int, int32	<int | xint> <int | xint>l	3, 0x3 3l, 0x3l	3
uint32	<int | xint>u <int | xint>ul	3u 3ul	3
byte, uint8	<int | xint>uy '<char>'B	97uy 'a'B	97
byte[]	"<string>"B @"<string>"B	"a\n"B @"a\n"B	[|97uy; 10uy|] [|97uy; 92uy; 110uy|]
sbyte, int8	<int | xint>y	3y	3
int16	<int | xint>s	3s	3
uint16	<int | xint>us	3us	3
int64	<int | xint>L	3L	3
uint64	<int | xint>UL <int | xint>uL	3UL 3uL	3
float, double	<float> <xint>LF	3.0 0x013LF	3.0 9.387247271e-323
single, float32	<float>F <float>f <xint>lf	3.0F 3.0f 0x013lf	3.0 3.0 4.4701421e-43f
decimal	<float | int>M <float | int>m	3.0M, 3M 3.0m, 3m	3.0
string	"<string>" @"<string>" """<string>"""	"\"quote\".\n" @"""quote"".\n" """"quote".\n"""	"quote".<newline> "quote".\n. "quote".\n

Table 3.3 List of literal types. The syntax notation <> means that the programmer replaces the brackets and content with a value of the appropriate form. The <xint> is one of the integers on hexadecimal, octal, or binary forms such as 0x17, 0o21, and 0b10001. The [| |] brackets means that the value is an array, see Section 13.4 for details.

The literal type is closely connected to how the values are represented internally. For example, a value of type `int32` uses 32 bits and can be both positive and negative, while a `uint32` value also uses 32 bits, but is unsigned. A `byte` is an 8-bit number, and `sbyte` is a signed 8-bit number. Values of type `float` use 64 bits, while `float32` only uses 32 bits. The number of bits used to represent numbers directly relates to the

range and precision these types can represent. This is summarized in Table 3.1 and discussed in more detail in Appendix B. String literals may be *verbatim* by the @-notation or triple double quotation marks, meaning that the escape sequences are not converted to their code point. The two types of string verbatim treat quotation marks differently, as illustrated in the table. Further examples are shown in Listing 3.4.

Listing 3.4: 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 = 7.900000095f

> 'A';;
val it: char = 'A'

> 'B'B;;
val it: byte = 66uy

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

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

Many basic types are compatible, and the type of a literal may be changed by *typecasting*. An example of casting to a `float` is shown in Listing 3.5. When `float`

Listing 3.5: Casting an integer to a floating point number.

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

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 4. Boolean values are often treated as integer values 0 and 1, but no short-hand function names exist for their conversions. Instead, use functions from the `System.Convert` family of functions, as demonstrated in Listing 3.6. 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 will be discussed in Chapter 9.

Typecasting is often a destructive operation, e.g., typecasting a `float` to `int` removes the fractional part without rounding as shown in Listing 3.7. Here we typecasted to

Listing 3.6: 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
```

Listing 3.7: Fractional part is removed by downcasting.

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

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 3.5 showed. Since floating point numbers are a superset of integers, the value is retained. 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 those in $y.x \in [y.5, y+1)$ to $y+1$. This can be performed by downcasting, as shown in Listing 3.8. I.e., $357.6 + 0.5 = 358.1$

Listing 3.8: Rounding by modified downcasting.

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

and removing the fractional part by downcasting results in 358, which is the correct answer.

3.2 Operators on Basic Types

Expressions are the basic building block of all F# programs, and this section will discuss operator expressions on basic types. A typical calculation, such used in Listing 3.8, is

$$\underbrace{357.6}_{\text{operand}} \quad \underbrace{+}_{\text{operator}} \quad \underbrace{0.5}_{\text{operand}} \tag{3.1}$$

is an example of an arithmetic *expression*, and the above expression consists of two *operands* and an *operator*. Since this operator takes two operands, it is called a

binary operator. The expression is written using *infix notation*, since the operands appear on each side of the operator.

In order to discuss general programming structures, we will use simplified language to describe valid syntactical structures. In this simplified language, the syntax of basic binary operators is shown in the following.

Listing 3.9: Syntax for a binary expression.

```
1 <expr><op><expr>
```

Here <expr> is any expression supplied by the programmer, and <op> is a binary infix operator. F# supports a range of arithmetic binary infix operators on its built-in types, such as addition, subtraction, multiplication, division, and exponentiation, using the "+", "-", "*", "/", "**" lexemes, 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 3.4 and 3.5, and a range of mathematical functions is shown in Table 3.6. Note that

Operator	bool	ints	floats	char	string	Example	Result	Description
+		✓	✓	✓	✓	`5 + 2`	`7`	Addition
-		✓	✓			`5.0 - 2.0`	`3.0`	Subtraction
*		✓	✓			`5 * 2`	`10`	Multiplication
/		✓	✓			`5.0 / 2.0`	`2.5`	Division
%		✓	✓			`5 % 2`	`1`	Remainder
**			✓			`5.0 ** 2.0`	`25.0`	Exponentiation
+		✓	✓			`+3`	`3`	identity
-		✓	✓			`-3.0`	`-3.0`	negation
&&	✓					`true && false`	`false`	boolean and
||	✓					`true || false`	`true`	boolean or
not	✓					`not true`	`false`	boolean negation
&&&		✓				`0b101 &&& 0b110`	`0b100`	bitwise boolean and
|||		✓				`0b101 ||| 0b110`	`0b111`	bitwise boolean or
^^^		✓				`0b101 ^^^ 0b110`	`0b011`	bitwise boolean exclusive or
<<<		✓				`0b110uy <<< 2`	`0b11000uy`	bitwise left shift
>>>		✓				`0b110uy >>> 2`	`0b1uy`	bitwise right shift
~~		✓				`~~~0b110uy`	`0b11111001uy`	bitwise boolean negation

Table 3.4 Arithmetic operators on basic types. Ints and floats means all built-in integer and float types. Note that for the bitwise operations, digits `0` and `1` are taken to be `true` and `false`.

expressions can themselves be arguments to expressions, and thus, `4+5+6` is also a legal statement. Technically, F# interprets the expression as `(4+5)+6` meaning that first `4+5` is evaluated according the <expr><op><expr> syntax. Then the result replaces the parenthesis to yield `9+6`, which, once again, is evaluated according to the <expr><op><expr> syntax to give `15`. This is called *recursion*, which is the name for a type of rule or function that uses itself in its definition. See Chapter 8 for more on recursive functions.

Operator	bool	ints	floats	char	string	Example	Result	Description
<	✓	✓	✓	✓	✓	`true < false`	`false`	Less than
>	✓	✓	✓	✓	✓	`5 > 2`	`true`	Greater than
=	✓	✓	✓	✓	✓	`5.0 = 2.0`	`false`	Equal
<=	✓	✓	✓	✓	✓	`'a' <= 'b'`	`true`	Less than or equal
>=	✓	✓	✓	✓	✓	`"ab" >= "cd"`	`false`	Greater than or equal
<>	✓	✓	✓	✓	✓	`5 <> 2`	`true`	Not equal

Table 3.5 Comparison operators on basic types. Types cannot be mixed, e.g., `3<'a'` is a syntax error.

Operator	bool	ints	floats	char	string	Example	Result	Description
`abs`		✓	✓			`abs -3`	`3`	Absolute value
`acos`			✓			`acos 0.8`	`0.644`	Inverse cosine
`asin`			✓			`asin 0.8`	`0.927`	Inverse sinus
`atan`			✓			`atan 0.8`	`0.675`	Inverse tangent
`atan2`			✓			`atan2 0.8 2.3`	`0.335`	Inverse tangentvariant
`ceil`			✓			`ceil 0.8`	`1.0`	Ceiling
`cos`			✓			`cos 0.8`	`0.697`	Cosine
`exp`			✓			`exp 0.8`	`2.23`	Natural exponent
`floor`			✓			`floor 0.8`	`0.0`	Floor
`log`			✓			`log 0.8`	`-0.223`	Natural logarithm
`log10`			✓			`log10 0.8`	`-0.0969`	Base-10 logarithm
`max`		✓	✓	✓	✓	`max 3.0 4.0`	`4.0`	Maximum
`min`		✓	✓	✓	✓	`min 3.0 4.0`	`3.0`	Minimum
`pown`		✓				`pown 3 2`	`9`	Integer exponent
`round`			✓			`round 0.8`	`1.0`	Rounding
`sign`		✓	✓			`sign -3`	`-1`	Sign
`sin`			✓			`sin 0.8`	`0.717`	Sinus
`sqrt`			✓			`sqrt 0.8`	`0.894`	Square root
`tan`			✓			`tan 0.8`	`1.03`	Tangent

Table 3.6 Predefined functions for arithmetic operations.

Unary operators take only one argument and have the syntax:

Listing 3.10: A unary expressions.

```
1 <op><expr>
```

An example of a unary operator is `-3`, where `-` here is used to negate a positive integer. Since the operator appears before the operand, it is a *prefix operator*.

The concept of *precedence* is an important concept in arithmetic expressions. If parentheses are omitted in Listing 3.8, then F# will interpret the expression as `(int 357.6) + 0.5`, which is erroneous since the addition of an integer with a float is undefined. This is an example of precedence, i.e., function evaluation takes precedence over addition which means that function evaluation is performed first and

addition second. Consider the arithmetic expression shown in Listing 3.11. Here, the

Listing 3.11: A simple arithmetic expression.

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

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)` and `(3 + 4) * 5` that gives different results. For integer arithmetic, the correct order is, of course, multiplication before addition, and we say that multiplication takes *precedence* over addition. Every atomic operation that F# can perform is ordered in terms of its precedence, and for some common built-in operators shown in Table 3.7, the precedence is shown by the order they are given in the table.

Operator	Associativity	Description

`+<expr>`
`-<expr>`
`~~~<expr>`
	Left	Unary identity, negation, and bitwise negation operators

`f <expr>`	Left	Function application
`<expr> ** <expr>`	Right	Exponentiation

`<expr> * <expr>`
`<expr> / <expr>`
`<expr> % <expr>`
	Left	Multiplication, division and remainder

`<expr> + <expr>` `<expr> - <expr>`	Left	Addition and subtraction binary operators
`<expr> ^^^ <expr>`	Right	Bitwise exclusive or

`<expr> < <expr>`
`<expr> <= <expr>`
`<expr> > <expr>`
`<expr> >= <expr>`
`<expr> = <expr>`
`<expr> <> <expr>`
`<expr> <<< <expr>`
`<expr> >>> <expr>`
`<expr> &&& <expr>`
`<expr> ||| <expr>`
	Left	Comparison operators, bitwise shift, and bitwise 'and' and 'or'.

`<expr> && <expr>`	Left	Boolean and
`<expr> || <expr>`	Left	Boolean or

Table 3.7 Some common operators, their precedence, and their associativity. Rows are ordered from highest to lowest precedences, such that `<*expr*> * <*expr*>` has higher precedence than `<*expr*> + <*expr*>`. Operators in the same row have the same precedence..

Associativity describes the order in which calculations are performed for binary operators of the same precedence. Some operator's associativity are given in Table 3.7. In the table we see that "*" is left associative, which means that `3.0 * 4.0 * 5.0` is evaluated as `(3.0 * 4.0) * 5.0`. Conversely, ** is right-associative, so `4.0**3.0**2.0` is evaluated as `4.0**(3.0**2.0)`. For some operators, like multiplication, association matters little, e.g., $4 * 3 * 2 = 4 * (3 * 2) = (4 * 3) * 2$, and for other operators, like exponentiation, the association makes a huge difference, e.g.,

★ $4^{(3^2)} \neq (4^3)^2$. Examples of this are shown in Listing 3.12. **Whenever in doubt of**

Listing 3.12: Precedence rules define implicit parentheses.

```
> 4.0 * 3.0 * 2.0;;
val it: float = 24.0

> (4.0 * 3.0) * 2.0;;
val it: float = 24.0

> 4.0 * (3.0 * 2.0);;
val it: float = 24.0

> 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
```

association or any other basic semantic rules, it is a good idea to use parentheses. It is also a good idea to test your understanding of the syntax and semantic rules by making a simple script.

3.3 Boolean Arithmetic

Boolean arithmetic is the basis of almost all computers and is particularly important for controlling program flow, which will be discussed in Section 13.2. 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# are written respectively as the binary operators `&&`, `||`, and the function `not`. Since the domain of Boolean values is so small, all possible combinations of input on these values can be written on the tabular form, known as a *truth table*, and the truth tables for the basic Boolean operators and functions are shown in Table 3.8. A good mnemonic for remembering the result of the 'and' and 'or' operators is to

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

use 1 for true, 0 for false, multiplication for the Boolean 'and' operator, and addition for the Boolean 'or' operator, e.g., true and false in this mnemonic translates to

$1 \cdot 0 = 0$, and the result translates back to the Boolean value false. In F#, the truth table for the basic Boolean operators can be produced by a program, as shown in Listing 3.13. Here, we used the `printfn` function to present the results of many

Listing 3.13: Boolean operators and truth tables.

```
1
2  > printfn "a b a*b a+b not a"
3  printfn "%A %A %A %A %A"
4    false false (false && false) (false || false) (not false)
5  printfn "%A %A %A %A %A"
6    false true (false && true) (false || true) (not false)
7  printfn "%A %A %A %A %A"
8    true false (true && false) (true || false) (not true)
9  printfn "%A %A %A %A %A"
10   true true (true && true) (true || true) (not true);;
11 a b a*b a+b not a
12 false false false false true
13 false true false true true
14 true false false true false
15 true true true true false
16 val it: unit = ()
```

expressions on something that resembles a tabular form. The spacing produced using the `printfn` function is not elegant, and in Chapter 12 we will discuss better options for producing more beautiful output. Notice that the arguments for `printfn` were 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 lightweight syntax. The difference between verbose and lightweight syntax is discussed in Chapter 4.

3.4 Integer Arithmetic

The set of integers is infinitely large, but since all computers have limited resources, it is not possible to represent it in its entirety. The various integer types listed in Table 3.1 are finite subsets reduced by limiting their ranges. An in-depth description of integer implementation can be found in Appendix B. The type `int` is the most common type.

Table 3.4–3.6 give examples of 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 is straightforward. However, performing arithmetic operations on integers requires extra care, since the result may cause *overflow* and *underflow*. For example, an `sbyte` is speci-

fied using the "y"-literal and can hold values $[-128 \ldots 127]$. This causes problems in the example in Listing 3.14. Here $100+30 = 130$, which is larger than the biggest

Listing 3.14: Adding integers may cause overflow.

```
> 100y;;
val it: sbyte = 100y

> 30y;;
val it: sbyte = 30y

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

`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`, as demonstrated in Listing 3.15. I.e., we were expecting a negative number but got a positive number

Listing 3.15: Subtracting integers may cause underflow.

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

instead.

The overflow error in Listing 3.14 can be understood in terms of the binary representation of integers: In binary, $130 = 10000010_2$, and this binary pattern is interpreted differently as `byte` and `sbyte`, see Listing 3.16. That is, for signed bytes, the left-

Listing 3.16: The leftmost 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
```

most bit is used to represent the sign, and since the addition of $100 = 01100100_2$ and $30 = 00011110_b$ is $130 = 10000010_2$, which causes the left-most bit to be used, this is wrongly interpreted as a negative number when stored in an `sbyte`. Similar arguments can be made explaining underflows.

The operator discards the fractional part after division, and the *integer remainder* operator calculates the remainder after integer division, as demonstrated in Listing 3.17. Together, the integer division and remainder can form a lossless representation of the original number, see Listing 3.18. Here we see that integer division of 7 by 3 followed by multiplication by 3 is less than 7, and that 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 by 0 is infinite,

Listing 3.17: Integer division and remainder operators.

```
1
2 > 7 / 3;;
3 val it: int = 2
4
5 > 7 % 3;;
6 val it: int = 1
```

Listing 3.18: Integer division and remainder is a lossless representation of an integer, compare with Listing 3.17.

```
1 > (7 / 3) * 3;;
2 val it: int = 6
3
4 > (7 / 3) * 3 + (7 % 3);;
5 val it: int = 7
```

which is also outside the domain of any of the integer types, but in this case, F# casts an *exception*, as shown in Listing 3.19. The output looks daunting at first sight,

Listing 3.19: Integer division by zero causes an exception runtime error.

```
1 > 3/0;;
2 System.DivideByZeroException: Attempted to divide by zero.
3    at <StartupCode$FSI_0002>.$FSI_0002.main@()
```

but the first and last lines of the error message are the most important parts, which tell us what exception was cast and why the program stopped. The middle contains technical details concerning which part of the program caused the error and can be ignored for the time being. Exceptions are a type of *runtime error*, and are discussed in Section 12.3

Integer exponentiation is not defined as an operator but is available as the built-in function `pown`. This function is demonstrated in Listing 3.20 for calculating 2^5.

Listing 3.20: Integer exponent function.

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

For binary arithmetic on integers, the following operators are available: `<leftExpr> <<< <rightExpr>`, which shifts the bit pattern of `<leftExpr>` `<rightExpr>` positions to the left while inserting 0's to right; `<leftExpr> >>> <rightExpr>`, which shifts the bit pattern of `<leftExpr>` `<rightExpr>` positions to the right while inserting 0's to left; `~~~ <expr>` returns a new integer, where all 0 bits are changed to 1 bits and vice-versa; `<expr> &&& <expr>` returns the result of taking the Boolean 'and' operator position-wise; `<expr> ||| <expr>` returns the result of taking the Boolean 'or' operator position-wise; and `<expr> ^^^ <expr>` returns the result of the Boolean 'xor' operator defined by the truth table in Table 3.9.

a	b	a ^^^ b
false	false	false
false	true	true
true	false	true
true	true	false

Table 3.9 Boolean exclusive or truth table.

3.5 Floating Point Arithmetic

Like integers, the set of reals is also infinitely large, hence, 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 B. The type `float` is the most common type.

Table 3.4–3.6 give examples of operators and functions pre-defined for floating point types. Note that the remainder operator for floats calculates the remainder after division and discards the fractional part, see Listing 3.21. The remainder for

Listing 3.21: Floating point division and remainder operators.

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

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

floating point numbers can be fractional, but division, discarding fractional part, and the remainder is still a lossless representation of the original number, as demonstrated in Listing 3.22.

Listing 3.22: Floating point division, downcasting, 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. As shown in Listing 3.23, no exception is thrown. 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., addition and subtraction can give surprising results, as demonstrated in Listing 3.24. 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. The

Listing 3.23: 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
```

Listing 3.24: Floating point arithmetic has finite precision.

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

reason is that the calculation is done stepwise, and in the process, the numbers are represented using the imprecise floating point standard. Thus,`357.8 + 0.1` is represented as a number close to but not identical to what `357.9` is represented as, and thus, when subtracting these two representations, we get a very small nonzero number. 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`.** ⋆

3.6 Char and String Arithmetic

Addition is the only operator defined for characters. Nevertheless, character arithmetic is often done by casting to an integer. A typical example is the conversion of character case, e.g., to convert the lowercase character 'z' to uppercase. Here, we use the *ASCIIbetical order*, add the difference between any Basic Latin Block letters in upper- and lowercase as `integers`, and cast back to `char`, see Listing 3.25. I.e., the code point difference between the upper and lower case for any alphabetical

Listing 3.25: Converting case by casting and integer arithmetic.

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

character 'a' to 'z' is constant, hence we can change the 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 simplest is concatenation using the "+" operator, as demonstrated in Listing 3.26. 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

Listing 3.26: Example of string concatenation.

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

be indexed as using the `[]` notation. This is demonstrated in Listing 3.27. No-

Listing 3.27: 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"
```

tice that the first character has index 0, and to get the last character in a string, we use the string's *`Length` property*. A Property is an extra piece of information associated with a given value. This is done as shown in Listing 3.28. Since

Listing 3.28: String Length property and string indexing.

```
> "abcdefg".Length;;
val it: int = 7

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

index counting starts at 0, and since the string length is 7, the index of the last character is 6. There is a long list of built-in functions in `System.String` for working with strings, see `https://fsharp.github.io/fsharp-core-docs/reference/fsharp-core-stringmodule.html` for further details.

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 objects, classes, and methods, see Chapter 15.

Strings are compared letter by letter. For two strings to be equal, they must have the same length and all the letters must be identical. E.g., `"abs" = "absalon"` is false,

while `"abs" = "abs"` is true. The "<>" operator is the boolean negation of the "=" operator, e.g., `"abs" <> "absalon"` is true, while `"abs" <> "abs"` is false. For the "<" , "<=", ">", and ">=" operators, the strings are ordered alphabetically, such that `"abs" < "absalon" && "absalon" < "milk"` is true, that is, the "<" operator on two strings is true if the left operand should come before the right when sorting alphabetically. The algorithm for deciding the boolean value of `leftOp < rightOp` is as follows: we start by examining the first character, and if `leftOp[0]` and `rightOp[0]` are different, then `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 on to 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 shorter word is smaller than the longer word, e.g., `"abs" < "absalon"` is true, while `"abs" < "abs"` is false. The "<=", ">", and ">=" operators are defined in a similar manner.

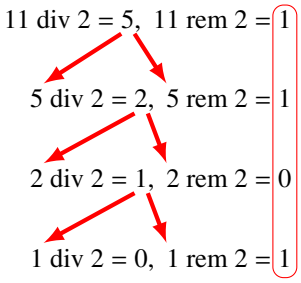
3.7 Programming Intermezzo: Hand Conversion Between Decimal and Binary Numbers

Conversion of integers between decimal and binary form is a key concept one must grasp in order to understand some of the basic properties of calculations on the computer. Converting from binary to decimal is straightforward if using the power-of-two algorithm, i.e., given a sequence of $n + 1$ binary digits b_i written as $b_n b_{n-1} \ldots b_0$, and where b_n and b_0 are the most and least significant bits respectively, then the decimal value is calculated as,

$$v = \sum_{i=0}^{n} b_i 2^i \tag{3.2}$$

For example, $10011_2 = 1 + 2 + 16 = 19$. Converting 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 dividing a number by two is equivalent to shifting its binary representation from one position to the right. E.g., $10 = 1010_2$ and $10/2 = 5 = 101_2$. 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 to say 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} in decimal form to binary form, we

would perform the following steps:



Here we used div and rem to signify the integer division and remainder operators. The algorithm 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 the interactive mode, we can perform the same calculation, as shown in Listing 3.29.

Listing 3.29: 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, by reading the second integer-response 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 algorithm may be used on the whole part, while multiply-by-two may be used in a similar manner on the fractional part.

3.8 Key Concepts and Terms in This Chapter

In this chapter you have learned about:

- the **basic types**: **int** of various kinds which are all a subset of integers, **float** of various kinds which are all subsets of reals, **bool** which captures the notion of true and false, and **char** and **string** which holds characters and sequences of characters;
- how to write constants of the basic types, which are called **literals**;
- **operators** such as "+" and "-" and whose arguments are called **operands**;
- **unary** and **binary** operators;
- **escape sequences** for characters and strings;
- how to get the **length** of a string using the **dot** notation, and how to extract substrings using the **slice** notation.