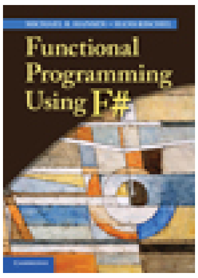

Functional Programming Using F#

Michael R. Hansen, Hans Rischel

Book DOI: http://dx.doi.org/10.1017/CBO9781139093996

Online ISBN: 9781139093996

Hardback ISBN: 9781107019027

Paperback ISBN: 9781107684065

Chapter

2 - Values, operators, expressions and functions pp. 21-42

Chapter DOI: http://dx.doi.org/10.1017/CBO9781139093996.003

Cambridge University Press

2

Values, operators, expressions and functions

The purpose of this chapter is to illustrate the use of values of basic types: numbers, characters, truth values and strings by means of some examples. The concepts of operator overloading and type inference are explained. Furthermore, the chapter contains a gentle introduction to higher-order functions. It is explained how to declare operators, and the concepts of equality and ordering in F# are introduced. After reading the chapter the reader should be able to construct simple programs using numbers, characters, strings and truth values.

2.1 Numbers. Truth values. The `unit` type

From mathematics we know the set of natural numbers as a subset of the set of integers, which again is a subset of the rational numbers (i.e., fractions), and so on. In F#, however, the set of values with the type: `int`, for the integers, is considered to be disjoint from the set of values with the type: `float`, for floating-point numbers, that is, the part of the real numbers that are representable in the computer. The reason is that the encodings of integer and float values in the computer are different, and that the computer has different machine instructions for adding integer values and for adding float values, for example.

A value of type `int` is written as a sequence of digits possibly prefixed with the minus sign "-". Real numbers are written using decimal point notation or using exponential notation, or using both:

```
0;;
val it : int = 0

0.0;;
val it : float = 0.0

0123;;
val it : int = 123

-7.235;;
val it : float = -7.235

-388890;;
val it : int = -388890
```

```
1.23e-17;;
val it : float = 1.23e-17
```

where `1.23e-17` denotes $1.23 \cdot 10^{-17}$.

Operators

We will use the term *operator* as a synonym for function and the components of the argument of an operator will be called *operands*. Furthermore, a *monadic* operator is an operator with one operand, while a *dyadic* operator has two operands. Most monadic operators are used in *prefix* notation where the operator is written in front of the operand.

Examples of operators on numbers are monadic minus `-`, and the dyadic operators addition `+`, subtraction `-`, multiplication `*` and division `/`. Furthermore, the relations: `=`, `<>` (denoting inequality $\neq$), `>`, `>=` (denoting $\geq$), `<` and `<=` (denoting $\leq$), between numbers are considered to be operators on numbers computing a truth value.

The symbol "–" is used for three purposes in F# as in mathematics. In number constants like "`-2`" it denotes the sign of the constant, in expressions like "`- 2`" and "`-(2+1)`" it denotes an application of the monadic minus operator, and in the expression "`1-2`" it denotes the dyadic subtraction operator.

Consider, as a strange example:

```
2 - - -1;;
val it : int = 1
```

Starting from the right, `-1` denotes the the integer "minus one ", the expression `- -1` denotes monadic minus applied to minus one, and the full expression denotes the dyadic operation two minus one.

Division is *not* defined on integers, but we have instead the operators `/` for quotient and `%` for remainder as described on Page 15, for example:

```
13 / -5;;
val it : int = -2

13 % -5;;
val it : int = 3
```

Truth values

There are two values `true` and `false` of the type `bool`:

```
true;;
val it : bool = true

false;;
val it : bool = false
```

Logical operators	
`not`	(unary) negation
`&&`	logical and (conjunction)
`||`	logical or (disjunction)

Table 2.1 *Operators on truth values*

Functions can have truth values as results. Consider, for example, a function `even` determining whether an integer n is even (i.e., n `%` $2 = 0$). This function can be declared as follows:

```
let even n = n % 2 = 0;;
val even : int -> bool
```

A truth-valued function such as `even` is called a *predicate*.

Functions on truth values are often called *logical operators*, and some of the main ones are shown in Table 2.1. The *negation* operator `not` applies to truth values, and the comparison operators = and <> are defined for truth values. For example:

```
not true <> false;;
val it : bool = false
```

Furthermore, there are expressions e_1 `||` e_2 and e_1 `&&` e_2 corresponding to the disjunction and conjunction operators of propositional logic. The expression e_1 `||` e_2 is true if either e_1 or e_2 (or both) are true; otherwise the expression is false. The expression e_1 `&&` e_2 is true if both e_1 and e_2 are true; otherwise the expression is false.

Evaluations of e_1 `||` e_2 and e_1 `&&` e_2 will only evaluate the expression e_2 when needed, that is, the expression e_2 in e_1 `||` e_2 is not evaluated if e_1 evaluates to `true`, and the expression e_2 in e_1 `&&` e_2 is not evaluated if e_1 evaluates to `false`. For example:

```
1 = 2 && fact -1 = 0;;
val it : bool = false
```

Thus, `1 = 2 && fact -1 = 0` evaluates to `false` without attempting to evaluate the expression `fact -1 = 0`, which would result in a non-terminating evaluation.

The `unit` *type*

There is only one value, written `()`, of type `unit`. It is mentioned here as it belongs to the basic types in F#. It is used in the imperative part of F# as a "dummy" result of a computation consisting solely of side-effects like input-output or modification of mutable data. There are no operators on the value `()` of type `unit`.

2.2 Operator precedence and association

The monadic operator − is written in front of the argument (like other function names), while the dyadic operators are written in *infix* notation, where the operator is placed between the operands. Table 2.2 shows the arithmetic operators.

`+`	unary plus
`-`	unary minus
`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`%`	modulo (remainder)
`**`	exponentiation

Table 2.2 *Arithmetic operators*

Usual rules for omitting brackets in mathematical expressions also apply to F# expressions. These rules are governed by two concepts: operator *precedence* and operator *association* for dyadic operators as shown in Table 2.3. The operators occurring in the same row have same precedence, which is higher than that of operators occurring in succeeding rows. For example, `*` and `/` have the same precedence. This precedence is higher than that of `+`.

Operator	Association
`**`	Associates to the right
`* / %`	Associates to the left
`+ -`	Associates to the left
`= <> > >= < <=`	No association
`&&`	Associates to the left
`||`	Associates to the left

Table 2.3 *Operator precedence and association*

Furthermore, a monadic operator (including function application) has higher precedence than any dyadic operator. The idea is that higher (larger) precedence means earlier evaluation. For example:

`- 2 - 5 * 7 > 3 - 1` means `((- 2) - (5*7)) > (3 - 1)`

and

`fact 2 - 4` means `(fact 2) - 4`

The dyadic operators for numbers and truth values (except `**`) *associate* to the *left*, which means that operators of the same precedence are applied starting from the left, so the evaluation of an expression will proceed as if the expression was fully bracketed. For example:

`1 - 2 - 3` means `(1 - 2) - 3`

2.3 Characters and strings

A character is a letter, a digit or a special character (i.e., a punctuation symbol like comma or semicolon or a control character). Characters are encoded in the computer as integer values using the *Unicode alphabet*, which is an international standard for encoding characters.

A character value is written as the character *c* enclosed in apostrophes. Examples of values of type `char` are:

```
'a';;
val it : char = 'a'

' ';;
val it : char = ' '
```

where the last one denotes the space character.

The new line, apostrophe, quote and backslash characters are written by means of the *escape sequences* shown in Table 2.4. Functions on characters are found in the `System.Char` library.

Sequence	Meaning
`\'`	Apostrophe
`\"`	Quote
`\\`	Backslash
`\b`	Backspace
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab

Table 2.4 *Character escape sequences*

The operators `||`, `&&` and `not` are convenient when declaring functions with results of type `bool`, like in the following declarations of the functions `isLowerCaseConsonant` and `isLowerCaseVowel` determining whether a character is a lower-case consonant or vowel, respectively:

```
let isLowerCaseVowel ch =
    ch='a' || ch='e' || ch='i' || ch='o' || ch='u';;
val isLowerCaseVowel : char -> bool

let isLowerCaseConsonant ch =
    System.Char.IsLower ch && not (isLowerCaseVowel ch);;
val isLowerCaseConsonant : char -> bool

isLowerCaseVowel 'i' && not (isLowerCaseConsonant 'i');;
val it : bool = true

isLowerCaseVowel 'I' || isLowerCaseConsonant 'I';;
val it : bool = false

not (isLowerCaseVowel 'z') && isLowerCaseConsonant 'z';;
val it : bool = true
```

where we use the function `IsLower` from the library `System.Char` to check whether `ch` is a lower-case letter. This library contains predicates `IsDigit`, `IsSeparator`, and so on, expressing properties of a character.

Strings

A *string* is a sequence of *characters*. Strings are values of the type `string`. A string is written inside enclosing quotes that are *not* part of the string. Quote, backslash or control characters in a string are written by using the escape sequences. Comments cannot occur inside strings as comment brackets (`(*` or `*)`) inside a string simply are interpreted as parts of the string. Examples of values of type `string` are:

```
"abcd---";;
val it : string = "abcd---"

"\"1234\"";;
val it : string = "\"1234\""

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

The first one denotes the 7-character string "`abcd---`", the second uses escape sequences to get the 6-character string "`"1234"`" including the quotes, while the last denotes the *empty string* containing no characters.

Strings can also be written using the *verbatim* string notation where the character `@` is placed in front of the first quote:

`@"`$c_0\ c_1\ \ldots\ c_{n-1}$`"`

It denotes the string of characters $c_0\ c_1\ \ldots\ c_{n-1}$ *without* any conversion of escape sequences. Hence `@"\\\\"` denotes a string of four backslash characters:

```
@"\\\\";;
val it : string = "\\\\"
```

while the escape sequence `\\` for backslash is converted in the string `"\\\\"`:

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

Verbatim strings are useful when making strings containing backslash characters. Note that it is not possible to make a verbatim string containing a quote character because `\"` is interpreted as a backslash character followed by the terminating quote character.

Functions on strings

The `String` library contains a variety of functions on strings. In this section we will just illustrate the use of a few of them by some examples.

The `length` function computes the number of characters in a string:

```
String.length "1234";;
val it : int = 4

String.length "\"1234\"";;
val it : int = 6
```

```
String.length "";;   // size of the empty string
val it : int = 0
```

The concatenation function + joins two strings together forming a new string by placing the two strings one after another. The operator + is used in infix mode:

```
let text = "abcd---";;
val text : string = "abcd---"

text + text;;
val it: string = "abcd---abcd---"

text + " " = text;;
val it : bool = false

text + "" = text;;
val it : bool = true

"" + text = text;;
val it : bool = true
```

The last two examples show that the empty string is the *neutral element* for concatenation of strings just like the number 0 is the neutral element for addition of integers.

Note that the *same* operator symbol + is used for integer addition and string concatenation. This *overloading* of operator symbols is treated in Section 2.5.

A string s with length n is given by a sequence of n characters $s =$ "$c_0 c_1 \cdots c_{n-1}$", where the convention in F# is that the numbering starts at 0. For any such string s there is a function, written $s.[i]$, to extract the i'th character in s for $0 \leq i \leq n-1$. The integer i used in $s.[i]$ is called an *index*. For example:

```
"abc".[0];;
val it : char = 'a'

"abc".[2];;
val it : char = 'c'

"abc".[3];;
System.IndexOutOfRangeException: ...
Stopped due to error
```

where the last example shows (a part of) the error message which will occur when the index is out of bounds.

If we want to concatenate a string and a character, we need to use the `string` function to convert the character to a string, for example

```
"abc" + string 'd';;
val it : string = "abcd"
```

as the operator + in this case denotes string concatenation, and this operator cannot concatenate a string with a character.

Conversion of integer, real or Boolean values to their string representations are done by using the function `string`, for example:

```
string -4;;
val it : string = "-4"

string 7.89;;
val it : string = "7.89"

string true;;
val it : string = "True"
```

A simple application of this conversion function is the declaration of the function `nameAge`:

```
let nameAge(name,age) =
    name + " is " + (string age) + " years old";;
```

It converts the integer value of the age to the corresponding string of digits and builds a string containing the string for the name and the age. For example:

```
nameAge("Diana",15+4);;
val it : string = "Diana is 19 years old"

nameAge("Philip",1-4);;
val it : string = "Philip is -3 years old"
```

The `string` function can actually give a string representation of every value, including values belonging to user-defined types. We shall return to this in Section 7.7. Examples of string representations are:

```
string (12, 'a');;
val it : string = "(12, a)"

string nameAge;;
val it : string = "FSI_0022+it@29-4"
```

where the pair `(12, 'a')` has a natural string representation in contrast to that of the user-defined `nameAge` function.

2.4 If-then-else expressions

An `if-then-else` expression has form:

`if` exp_1 `then` exp_2 `else` exp_3

where exp_1 is an expression of type `bool` while exp_2 and exp_3 are expressions of the same type. The `if-then-else` expression is evaluated by first evaluating exp_1. If exp_1 evaluates to `true` then the expression exp_2 is evaluated; otherwise, if exp_1 evaluates to `false` then the expression exp_3 is evaluated. Note that at most one of the expressions exp_2

and exp_3 will be evaluated (none of them will be evaluated if the evaluation of exp_1 does not terminate).

An `if-then-else` expression is used whenever one has to express a splitting into cases that cannot be expressed conveniently by use of patterns. As an example we may declare a function on strings that adjusts a string to even size by putting a space character in front of the string if the size is odd. Using the function `even` on Page 23 and `if-then-else` for the splitting into cases gives the following declaration:

```
let even n = n % 2 = 0;;
val even : int -> bool

let adjString s = if even(String.length s)
                  then s else " " + s;;
val adjString : string -> string

adjString "123";;
val it : string = " 123"

adjString "1234";;
val it : string = "1234"
```

One may, of course, use an `if-then-else` expression instead of splitting into clauses by pattern matching. But pattern matching is to be preferred, as illustrated by the following (less readable) alternative declaration of the `gcd` function (cf. Page 16):

```
let rec gcd(m,n) = if m=0 then n
                   else gcd(n % m,m);;
val gcd : int * int -> int
```

One should also avoid expressions of the forms:

`if` e_1 `then true else` e_2
`if` e_1 `then` e_2 `else false`

for defining Boolean combinations of expressions and instead use the shorter, equivalent forms:

e_1 `||` e_2
e_1 `&&` e_2

2.5 Overloaded functions and operators

A name or symbol for a function or operator is *overloaded* if it has different meanings when applied to arguments or operands of different types. We have already seen that the plus operator + denote addition for integers but concatenation for strings.

A (mathematical) function on real numbers is considered different from the corresponding function on integers, as they are implemented in F# by different machine instructions. An operator of this kind is hence overloaded: it denotes different functions depending on the context, and it depends on the types of the operands whether, for example, the operator `*` denotes multiplication on integers (of type `int`) or multiplication on real numbers (of type `float`). The F# system tries to resolve these ambiguities in the following way:

- If the type can be inferred from the context, then an overloaded operator symbol is interpreted as denoting the function on the inferred type.
- If the type cannot be inferred from the context, then an overloaded operator symbol with a default type will default to this type. The default type is `int` if the operator can be applied to integers.

For example, the obvious declaration of a squaring function yields the function on integers:

```
let square x = x * x;;
val square : int -> int
```

Declaring a squaring function on reals can be done either by specifying the type of the argument:

```
let square (x:float) = x * x;;
val square : float -> float
```

or by specifying the type of the result:

```
let square x : float = x * x;;
val square : float -> float
```

or by specifying the type of the expression for the function value:

```
let square x = x * x : float;;
val square : float -> float
```

or by choosing any mixture of the above possibilities.

`abs, acos, atan, atan2, ceil, cos, cosh, exp, floor, log`
`log10, pow, pown, round, sin, sinh, sqrt, tan, tanh`

Table 2.5 *Mathematical functions*

There are many overloaded operators in F#, in particular mathematical functions that can be applied to integers as well as to real numbers. Some of them can be found in Table 2.5. The function `abs`, for example, computes the absolute value of a number that can be of type `int`, `float` or any of the number types in Table 2.6, for example, `float32`:

```
abs -1;;
val it : int = 1

abs -1.0;;
val it : float = 1.0

abs -3.2f;;
val it : float32 = 3.20000000f
```

Overloading is extensively used in the .NET library and typing of arguments is frequently needed to resolve ambiguities. The user may declare overloaded operators and functions inside a type declaration as explained in Section 7.3.

2.6 Type inference

When an expression is entered, the F# system will try to determine a unique type using so-called *type inference*. If this does not succeed then the expression is not accepted and an error message is issued.

Consider once more the declaration of the function `power` (cf. Section 1.5):

```
let rec power = function
    | (x, 0) -> 1.0                  (* 1 *)
    | (x, n) -> x * power(x,n-1)     (* 2 *);;
val power : float * int -> float
```

The F# system deduces that `power` has the type: `float * int -> float`. We can see how F# is able to infer this type of `power` by arguing as follows:

1. The keyword `function` indicates that the type of power is a function type τ `->` τ', for some types τ and τ'.
2. Since `power` is applied to a pair `(x,n)` in the declaration, the type τ must have the form $\tau_1 * \tau_2$ for some types τ_1 and τ_2.
3. We have $\tau_2 =$ `int`, since the pattern of the first clause is `(x,0)`, and `0` has type `int`.
4. We have that $\tau' =$ `float`, since the expression for the function value in the first clause: `1.0` has type `float`.
5. We know that `power(x,n-1)` has the type `float` since $\tau' =$ `float`. Thus, the overloaded operator symbol `*` in `x * power(x,n-1)` resolves to `float` multiplication and `x` must be of type `float`. We hence get $\tau_1 =$ `float`.

The above declaration of the power function has been used for illustrating the declaration of recursive functions and the type inference performed by the system. As described above there is already a power operator `**` in F# and this should of course be used in programs. In general we recommend to inspect the F# and .NET libraries and use available library functions when appropriate.

2.7 Functions are first-class citizens

In functional languages, and F# is no exception, functions are what is called *first-class citizens*. An implication of this is that a function can be argument of another function and that the value of a function can again be a function. In this section we shall give a first, gentle introduction to this concept, which also is known as *higher-order* functions.

The value of a function can be a function

As a first example we shall consider the infix operator +. There is a version of this operator that is not written between the operands. This *non-fix* version is written `(+)`, and we shall now study its type:

```
(+);;
val it : (int -> int -> int) = <fun:it@1>
```

The type operator "->" *associates* to the *right*, so `(+)` has the type:

```
(+) : int -> (int -> int)
```

This type shows that the value of the function `(+)` is another function with type `int -> int`. Applying `(+)` to an integer n thus gives a function:

$$\texttt{(+)}\ n\texttt{: int -> int}$$

For example:

```
let plusThree = (+) 3;;
val plusThree : (int -> int)

plusThree 5;;
val it : int = 8

plusThree -7;;
val it : int = -4
```

The sum of two integers m and n can be computed as $(\texttt{(+)}\ m)\,n$. The brackets can be omitted because *function application associates to the left*. For example:

```
(+) 1 3;;
val it : int = 4
```

The argument of a function can be a function

Function composition $f \circ g$ is defined in mathematics by: $(f \circ g)(x) = f(g(x))$. This operator on functions is well-defined when domains and ranges of f and g match:

If $f : A \rightarrow B$ and $g : C \rightarrow A$, then $f \circ g : C \rightarrow B$

For example, if $f(y) = y + 3$ and $g(x) = x^2$, then $(f \circ\ g)(z) = z^2 + 3$.

We want to construe the function composition $\circ$ as a function, and this function will obviously take functions as arguments. There is actually an infix operator << in F# denoting function composition, and the above example can hence be paraphrased as follows:

```
let f = fun y -> y+3;;          // f(y) = y+3
val f : int -> int

let g = fun x -> x*x;;          // g(x) = x*x
val g : int -> int
```

```
let h = f << g;;                 // h = (f o g)
val h : int -> int

h 4;;                            // h(4) = (f o g)(4)
val it : int = 19
```

Using function expressions instead of named functions f, g and h, the example looks as follows:

```
((fun y -> y+3) << (fun x -> x*x)) 4;;
val it : int = 19
```

Declaration of higher-order functions

So far we have seen higher-order built-in functions like `(+)` and `(<<)`. We shall now illustrate ways to declare such functions by means of a simple example.

Suppose that we have a cube with side length s, containing a liquid with density ρ. The weight of the liquid is then given by $\rho \cdot s^3$. If the unit of measure of ρ is kg/m^3 and the unit of measure of s is m then the unit of measure of the weight will be kg.

Consider the following declaration of the weight function:

```
let weight ro = fun s -> ro * s ** 3.0;;
val weight : float -> float -> float
```

where we use the operator `**` to compute x^y for floating-point numbers x and y. A function value `weight` ρ is again a *function* as the expression on the right-hand side of the declaration is a `fun`-expression. This property of the function value is also visible in the type of `weight`.

We can make *partial evaluations* of the function `weight` to define functions for computing the weight of a cube of either water or methanol (having the densities $1000\mathrm{kg}/\mathrm{m}^3$ and $786.5\mathrm{kg}/\mathrm{m}^3$ respectively under "normal" pressure and temperature):

```
let waterWeight = weight 1000.0;;
val waterWeight : (float -> float)

waterWeight 1.0;;
val it : float = 1000.0

waterWeight 2.0;;
val it : float = 8000.0

let methanolWeight = weight 786.5;;
val methanolWeight : (float -> float)

methanolWeight 1.0;;
val it : float = 786.5

methanolWeight 2.0;;
val it : float = 6292.0
```

Higher-order functions may alternatively be defined by supplying the arguments as follows in the `let`-declaration:

```
let weight ro s = ro * s ** 3.0;;
val weight : float -> float -> float
```

and this is normally the preferred way of defining higher-order functions.

2.8 Closures

A *closure* gives the means of explaining a value that is a function. A closure is a triple:

$$(x,\ exp,\ env)$$

where x is an argument identifier, *exp* is the expression to evaluate to get a function value, while *env* is an environment (cf. Section 1.7) giving bindings to be used in such an evaluation.

Consider as an example the evaluation of `weight 786.5` in the previous example. The result is the closure:

$$\left(\texttt{s},\ \texttt{ro*s**3.0},\ \begin{bmatrix} \texttt{ro} & \mapsto & 786.5 \\ \texttt{*} & \mapsto & \textit{"the product function"} \\ \texttt{**} & \mapsto & \textit{"the power function"} \end{bmatrix}\right)$$

The environment contains bindings of all identifiers in the expression `ro*s**3.0` except the argument `s`.

Note that a closure is a *value* in F# – functions are first-class citizens.

The following simple example illustrates the role of the environment in the closure:

```
let pi = System.Math.PI;;
let circleArea r = pi * r * r;;
val circleArea : float -> float
```

These declarations bind the identifier `pi` to a `float` value and `circleArea` to a closure:

$$\begin{array}{lll} \texttt{pi} & \mapsto & 3.14159\ldots \\ \texttt{circleArea} & \mapsto & (\texttt{r},\ \texttt{pi*r*r},\ [\texttt{pi} \mapsto 3.14159\ldots]) \end{array}$$

A fresh binding of `pi` does not affect the meaning of `circleArea` that uses the binding of `pi` in the closure:

```
let pi = 0;;
circleArea 1.0;;
val it : float = 3.141592654
```

This feature of F# is called *static binding* of identifers occurring in functions.

2.9 Declaring prefix and infix operators

Expressions containing functions on pairs can often be given a more readable form by using infix notation where the function symbol is written between the two components of the argument. Infix form is used for the dyadic arithmetic operators `+`, `-`, `%`, `/`, for example. This allows us to make expressions more readable by use of rules for omission of brackets. For example: $x-y-z$ means $(x-y)-z$ and $x+y\star z$ means $x+(y\star z)$. These rules for omitting brackets are governed by *precedence* and *association* for the operators (see Section 2.2).

Operators are written using special character strings which cannot be used as "normal" identifiers. Infix operators are sequences of the following symbols[1]

```
!  %  &  *  +  -  .  /  <  =  >  ?  @  ^  |  ~
```

while prefix operators are one of

```
+   -   +.   -.   &   &&   %   %%
~   ~~   ~~~   ~~~~
```
(tilde characters)

The *bracket notation* converts from infix or prefix operator to (prefix) function:

- The corresponding (prefix) function for an infix operator *op* is denoted by `(`*op*`)`.
- The corresponding (prefix) function for a prefix operator *op* is denoted by `(~`*op*`)`.

An infix operator is declared using the bracket notation as in the following declaration of an infix exclusive-or operator `.||.` on truth values:

```
let (.||.) p q = (p || q) && not(p && q);;
val ( .||. ) : bool -> bool -> bool

(1 > 2) .||. (2 + 3 < 5);;
val it : bool = false
```

The system determines the precedence and association of declared operators on the basis of the characters in the operator. In the case of `.||.` the periods have no influence on this, so the precedence and association of `.||.` will be the same as those of `||`. Therefore,

```
true .||. false && true;;
```

is equivalent to

```
true .||. (false && true);;
```

as `&&` has higher precedence than `||` and `.||.`.

A prefix operator is declared using a leading tilde character. We may, for example, declare a prefix operator `%%` to calculate the reciprocal value of a `float` as follows:

```
let (~%%) x = 1.0 / x;;
val ( ~%% ) : float -> float

%% 0.5;;
val it : float = 2.0
```

[1] This description of legal operators in F# is incomplete. The precise rules are complicated.

Remark: When defining an operator starting or ending in an asterisks "`*`" a space must be inserted after "`(`" or before "`)`" to avoid a conflict with the comment convention using "`(*`" and "`*)`".

2.10 Equality and ordering

The *equality* and *inequality* operators = and <> are defined on any basic type and on strings:

```
3.5 = 2e-3;;
val it : bool = false

"abc" <> "ab";;
val it : bool = true
```

It is not defined on functions (closures):

```
cos = sin;;
stdin(5,1): error FS0001: The type '( ^a ->  ^a) ...
does not support the 'equality' constraint because
it is a function type
```

No type containing a function type can support equality as F# has no means to decide whether two functions are equal: It is a fundamental fact of theoretical computer science that there exists no (always terminating) algorithm to determine whether two arbitrary programs `f` and `g` (i.e., two closures) denote the same function.

The equality function is automatically extended by F# whenever the user defines a new type – in so far as the type does not contain function types.

The type of the function `eqText` declared by:

```
let eqText x y =
    if x = y then "equal" else "not equal";;
val eqText : 'a -> 'a -> string when 'a : equality
```

contains a *type variable* `'a` with the *constraint*: `when 'a : equality`.

This means that `eqText` will accept parameters `x` and `y` of any type τ equipped with equality:

```
eqText 3 4;;
val it : string = "not equal"

eqText ' ' (char 32);;
val it : string = "equal"
```

Ordering

The *ordering* operators: >, >=, <, and <= are defined on values of basic types and on strings. They correspond to the usual ordering of numbers. The ordering of characters is given by the ordering of the Unicode values, while `true` > `false` in the ordering of truth values.

Strings are ordered in the *lexicographical* ordering. That is, for two strings s_1 and s_2 we have that $s_1 < s_2$ if s_1 would occur before s_2 in a lexicon. For example:

```
                 // Upper case letters precede
'A' < 'a';;      // lower case letters
val it : bool = true

"automobile" < "car";;
val it : bool = true

"" < " ";;
val it : bool = true
```

Thus, the empty string precedes the string containing a space character, and the empty string precedes any other string in the lexicographical ordering. Ordering is automatically extended by F# whenever the user defines a new type, in so far as the type does not contain functions.

Using the comparison operators one may declare functions on values of an arbitrary type equipped with an ordering:

```
let ordText x y = if x > y then "greater"
                  else if x = y then "equal"
                  else "less";;
val ordText : 'a -> 'a -> string when 'a : comparison
```

The type of `x` and `y` contains a *type variable* `'a` with the *constraint*

```
when 'a : comparison
```

indicating that `x` and `y` can be of any type equipped with an ordering.

The library function `compare` is defined such that:

$$\texttt{compare}\ x\ y \;=\; \begin{cases} >0 & \text{if } x > y \\ 0 & \text{if } x = y \\ <0 & \text{if } x < y \end{cases}$$

where the precise value of `compare x y` depends on the structure of the values `x` and `y`.

It may be convenient to use *pattern matching with guards* when declaring functions using the `compare` function, for instance:

```
let ordText x y = match compare x y with
                  | t when t > 0 -> "greater"
                  | 0            -> "equal"
                  | _            -> "less";;
val ordText : 'a -> 'a -> string when 'a : comparison
```

The *guard* "`when t > 0`" restricts the matching, while the pattern "`t`" would otherwise match any value.

Type	**Description**	**Constant**
`bool`	Logical value	`true`, `false`
`unit`	Void	`()`
`char`	Character	`'`*char*`'`
`byte`	8-bit unsigned integer	*digits* `uy` or `0x` *hexdigits* `uy`
`sbyte`	8-bit signed integer	{-}*digits* `y` or {-}`0x` *hexdigits* `y`
`int16`	16-bit signed integer	{-}*digits* `s` or {-}`0x` *hexdigits* `s`
`uint16`	16-bit unsigned integer	*digits* `us` or `0x` *hexdigits* `us`
`int` (or `int32`)	32-bit signed integer	{-}*digits* or {-}`0x` *hexdigits*
`uint32`	32-bit unsigned integer	*digits* `u` or `0x` *hexdigits* `u`
`int64`	64-bit signed integer	{-}*digits* `L` or {-}`0x` *hexdigits* `L`
`uint64`	64-bit unsigned integer	*digits* `UL` or `0x` *hexdigits* `UL`
`nativeint`	Machines integer	{-}*digits* `n` or {-}`0x` *hexdigits* `n`
`unativeint`	Machines unsigned integer	*digits* `un` or `0x` *hexdigits* `un`
`float32` (or `single`)	32-bit IEEE floating-point	{-} *digits* `.` *digits* `f` or {-}*digits*{`.` *digits*} `e` {-} *digits* `f`
`float` (or `double`)	64-bit IEEE floating-point	{-}*digits* `.` *digits* or {-}*digits*{`.` *digits*} `e`{-}*digits*
`decimal`	High-precision decimal	*digits* `M` or {-}*digits* `.` *digits* `M`
`bigint`	Arbitrary integer	{-}*digits* `I`
`bignum`	Arbitrary rational number	{-}*digits* `N`

Table 2.6 *Basic Types*

2.11 Function application operators `|>` and `<|`

The operator `|>` means "send the value as argument to the function on the right" while `<|` means "send the value as argument to the function on the left," that is:

$$\begin{array}{lcl} arg\ |\!>\ fct & \text{means} & fct\ arg \\ fct\ <\!|\ arg & \text{means} & fct\ arg \end{array}$$

These operators are sometimes useful to make expressions more readable. There are two reasons for that:

- The operator `|>` allows you to write the argument to the *left* of the function.
- The operators `|>` and `<|` have lower precedence than the arithmetic operators.

Both expressions `a+b |> sin` and `sin <| a+b` do hence mean `sin(a+b)`. The operator `|>` has precedence over `<|` so `2 |> (-) <| 3` means `(2 |> (-)) <| 3`.

Both operators associate to the left. The parentheses in `2 |> (3 |> (-))` are hence needed to get the rightmost `|>` operator applied before the leftmost.

2.12 Summary of the basic types

The F# system supports a number of basic types not addressed previously in this chapter. Table 2.6 depicts the basic types, where the column "Constant" describes how constants are written. The meta symbols *digits*, *hexdigits*, *char*, and {...} have the following meanings:

digits: One or more decimal digits: `0, 1, ..., 9` *hexdigits:* One or more hex digits: `0`, `1`, ..., `9`, `A`, `B`, ..., `F`, `a`, `b`, ..., `f`

char: A character or an escape sequence denoting a character.

{...}: The part between the brackets is optional. The brackets are not part of the string.

Hence `33e-8` is a constant of type `float` and `-0x1as` is a constant of type `int16` while `32f` is not accepted by F#.

Each type name denotes an overloaded conversion function converting to a value of the type in question (in so far as this is possible).

Summary

In this chapter we have described values and functions belonging to the basic F# types: integers, reals, characters, truth values and strings. Furthermore, we have discussed evaluation of infix operators with precedences, and the typing of arithmetic expressions where some operators may be overloaded. The concept of higher-order functions was introduced and the concept of a closure was used to explain the meaning of a function in F#. It was explained how to declare operators, and finally, the concepts of equality and ordering were explained.

Exercises

2.1 Declare a function `f: int -> bool` such that $\texttt{f}(n) = \texttt{true}$ exactly when n is divisible by 2 or divisible by 3 but not divisible by 5. Write down the expected values of `f`(24), `f`(27), `f`(29) and `f`(30) and compare with the result. Hint: n is divisible by q when $n\%q = 0$.

2.2 Declare an F# function `pow: string * int -> string`, where:

$$\texttt{pow}(s, n) = \underbrace{s \cdot s \cdot \cdots \cdot s}_{n}$$

where we use $\cdot$ to denote string concatenation. (The F# representation is `+`.)

2.3 Declare the F# function `isIthChar: string * int * char -> bool` where the value of $\texttt{isIthChar}(\mathit{str}, i, \mathit{ch})$ is true if and only if ch is the i'th character in the string str (numbering starting at zero).

2.4 Declare the F# function `occFromIth: string * int * char -> int` where

$$\texttt{occFromIth}(\mathit{str}, i, \mathit{ch}) \quad = \quad \begin{array}{l}\text{the number of occurrences of character } \mathit{ch} \\ \text{in positions } j \text{ in the string } \mathit{str} \text{ with } j \geq i.\end{array}$$

Hint: the value should be 0 for $i \geq \texttt{size}\ \mathit{str}$.

2.5 Declare the F# function `occInString: string * char -> int` where

$$\texttt{occInString}(\mathit{str}, \mathit{ch}) \quad = \quad \begin{array}{l}\text{the number of occurrences of character } \mathit{ch} \\ \text{in the string } \mathit{str}.\end{array}$$

2.6 Declare the F# function `notDivisible: int * int -> bool` where

$$\texttt{notDivisible}(d, n) \text{ is true if and only if } d \text{ is not a divisor of } n.$$

For example `notDivisible(2,5)` is true, and `notDivisible(3,9)` is false.

2.7 1. Declare the F# function `test: int * int * int -> bool`. The value of $\texttt{test}(a, b, c)$, for $a \leq b$, is the truth value of:

$$\begin{array}{ll} & \texttt{notDivisible}(a, c) \\ \text{and} & \texttt{notDivisible}(a+1, c) \\ \vdots & \\ \text{and} & \texttt{notDivisible}(b, c) \end{array}$$

2. Declare an F# function `prime: int -> bool`, where $\texttt{prime}(n) = \text{true}$, if and only if n is a prime number.
3. Declare an F# function `nextPrime: int -> int`, where $\texttt{nextPrime}(n)$ is the smallest prime number $> n$.

2.8 The following figure gives the first part of Pascal's triangle:

```
        1
      1   1
    1   2   1
  1   3   3   1
1   4   6   4   1
```

The entries of the triangle are called *binomial* coefficients. The k'th binomial coefficient of the n'th row is denoted $\binom{n}{k}$, for $n \geq 0$ and $0 \leq k \leq n$. For example, $\binom{2}{1} = 2$ and $\binom{4}{2} = 6$. The first and last binomial coefficients, that is, $\binom{n}{0}$ and $\binom{n}{n}$, of row n are both 1. A binomial coefficient inside a row is the sum of the two binomial coefficients immediately above it. These properties can be expressed as follows:

$$\binom{n}{0} = \binom{n}{n} = 1$$

and

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \text{ if } n \neq 0,\ k \neq 0, \text{ and } n > k.$$

Declare an F# function `bin: int * int -> int` to compute binomial coefficients.

2.9 Consider the declaration:

```
let rec f = function
  | (0,y) -> y
  | (x,y) -> f(x-1, x*y);;
```

1. Determine the type of `f`.
2. For which arguments does the evaluation of `f` terminate?
3. Write the evaluation steps for `f(2,3)`.
4. What is the mathematical meaning of $\texttt{f}(x, y)$?

2.10 Consider the following declaration:

```
let test(c,e) = if c then e else 0;;
```

1. What is the type of `test`?
2. What is the result of evaluating `test(false, fact(-1))`?
3. Compare this with the result of evaluating

```
if false then fact -1 else 0
```

2.11 Declare a function `VAT: int -> float -> float` such that the value `VAT n x` is obtained by increasing `x` by `n` percent.

Declare a function `unVAT: int -> float -> float` such that

```
unVAT n (VAT n x) = x
```

Hint: Use the conversion function `float` to convert an `int` value to a `float` value.

2.12 Declare a function `min` of type `(int -> int) -> int`. The value of `min`(f) is the smallest natural number n where $f(n) = 0$ (if it exists).

2.13 The functions `curry` and `uncurry` of types

```
curry   : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

are defined in the following way:

`curry` f is the function g where $g\ x$ is the function h where $h\ y = f(x, y)$.

`uncurry` g is the function f where $f(x, y)$ is the value $h\ y$ for the function $h = g\ x$.

Write declarations of `curry` and `uncurry`.