6 Values and functions

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 discriminant, $b^2 - 4ac > 0$. In order to write a program, where the code may be reused later, we define a function `discriminant : float -> float -> float -> float` , that is, a function that takes 3 arguments, `a`, `b`, and `c`, and calculates the discriminant. Details on function definition is given in Section 6.2. Likewise, we will define functions `positiveSolution : float -> float -> float -> float` and `negativeSolution : float -> float -> float -> float`, that also takes the polynomial's coefficients as arguments and calculates the solution corresponding to choosing the positive 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
do printfn "0 = %A * x ** 2.0 + %A * x + %A" a b c
do printfn "  has discriminant %A and solutions %A and %A" d xn xp
```

```
$ fsharpc --nologo identifiersExample.fsx && mono identifiersExample.exe
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 are 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 the amount of code, which needs to be debugged.

The use of identifiers is central in programming. For F# not to be confused by built-in functionality, identifiers must follow a specific set of rules:

· identifier

Identifier

- Identifiers are used as names for values, functions, types etc.
- They must start with a letter or underscore '_', but can be followed by zero or more of letters, digits, and a range of special characters except for SP, LF, and CR (space, line feed, and carriage return). Letters are most Unicode codepoints that produce regular letters, see Appendix C.3 for more on codepoints.
- They can also be a sequence of identifiers separated by a period.
- They cannot be keywords, see Table 6.1.

Examples of identifiers are: `a`, `theCharacter9`, `Next_Word`, `_tok`, and `f.sharp.rocks`. Since programmers often work in multilingual environment dominated by the English language **Restrict identifiers to use letters from the English alphabet, numbers, period, and '_'.** However, the number of possible identifiers is enormous: the full definition refers to the Unicode general categories described in Appendix C.3, and there are currently 19.345 possible Unicode code points in the letter category and 2.245 possible Unicode code points in the special character category.

Advice

Identifiers may be used to carry information about their intended content and use, and careful selection of identifiers can aid programmers to communicate thoughts about the code. Thus, identifiers are often a word or several concatenated words from the human language. For example in the function definition `let discriminant a b c = b ** 2.0 - 4.0 * a * c`, the function identifier has been chosen to be `discriminant`. F# places no special significance to the word 'discriminant', and the program would work exactly the same, had the function been called `let f a b c = b ** 2.0 - 4.0 * a * c`. However, to programmers, the word 'discriminant' informs us of the intended role of the function and

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

Table 6.1: Table of (possibly future) *keywords* and symbolic keywords in F#.

thus is much preferred. This is a general principle, **identifier names should be chosen to reflect their semantic value.**. The arguments `a`, `b`, and `c` are short, but adheres to a textbook tradition of elementary algebra. Again, we might as well have used, `let discriminant c a b = a ** 2.0 - 4.0 * c * b`, which is semantically identical to the original expression, but due to tradition, this would confuse most readers of the code. Thus, **identifier names should be chosen consistently with readers tradition.** Finally, identifiers are often concatenations of words, as `positiveSolution` in Listing 6.1. Concatenations can be difficult to read. Without the capitalized second word, we would have had `positivesolution`. This is readable at most times but takes longer time for humans to parse in general. Typical solutions are to use a separator such as `positive_solution`, *lower camel case* also known as *mixed case* as in the example `positiveSolution`, and *upper camel case* also known as *pascal case* as `PositiveSolution`. In this book, we use lower camel case except where F# requires a capital first letter. Again, the choice does not influence what a program does, only how readable it is to a fellow programmer. The important part is that **identifier names consisting of concatenated words are often preferred over few character names, and concatenation should be emphasized, e.g., by camel casing.** The length of identifier names is a balancing act, since when working with large programs, very long identifier names can be tiresome to write, and a common practice is that the length of identifier names is proportional to the complexity of the program. I.e., complex programs use long names, simple use short. What is complex and what is simple is naturally in the eye of the beholder, but when we program, remember that a future reader of the program most likely has not had time to work the problem as long as the programmer, thus **choose identifier names as if you were to explain the meaning of a program to a knowledgeable outsider.**

Advice
· lower camel case
· mixed case
· upper camel case
· pascal case
Advice
Advice

Another key concept in F# is expressions. 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 identifiers, which is done with the keyword `let`, e.g., `let a = 1.0`.

Expressions are the main workhorse of F# and have an enormous variety in how they may be written, we will in this book gradually work through some of the more important facets.

· expression

Expressions

- An Expression is a computation such as `3 * 5`.
- They can be value bindings between identifiers and expressions that evaluate to a value or a function, see Sections 6.1 and 6.2.
- They can be do bindings that produce side-effects and whose result is ignored, see Section 6.2
- They can be assignments to variables, see Section 6.1.

- They can be a sequence of expressions separated with "`;`" lexeme.
- They can be annotated with a type using the "`:`" lexeme. · :

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

· verbose syntax
· lightweight syntax

6.1 Value bindings

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`. Value bindings have the following syntax,

· value-binding

Listing 6.2 Value binding expression.

```
let <valueIdent> = <bodyExpr> [in <expr>]
```

The *`let`* keyword binds a value-identifier `<valueIdent>` with an expression `<bodyExpr>` that evaluates to a value. If the *`in`* keyword is used, then the value-identifier becomes synonymous with the evaluated value in `<expr>` only. The square bracket notation `[]` means that the enclosed is optional. Here the meaning is that for lightweight syntax, the `in` keyword is replaced with a newline, and the binding is valid in the following lines at the level of scope of the value-binding or deeper *lexically*.

· `let`
· `in`
· lexically

The value identifier annotated with a type using the "`:`" lexeme followed by the name of a type, e.g., `int`. The "`_`" lexeme may be used as a value-identifier. This lexeme is called the *wildcard* pattern, and for value-bindings it means that the `<bodyExpr>` is evaluated, but the result is discarded. See Chapter 15 for more details on patterns.

· :
· _
· wildcard

For example, letting the identifier `p` be bound to the value `2.0` and using it in an expression is done as shown in Listing 6.3.

Listing 6.3 letValue.fsx:
The identifier p is used in the expression following the in keyword.

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

```
$ fsharpc --nologo letValue.fsx && mono letValue.exe
9.0
```

F# will ignore most newlines between lexemes, i.e., the above is equivalent to writing as shown in Listing 6.4.

Listing 6.4 letValueLF.fsx:
Newlines after in make the program easier to read.

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

```
$ fsharpc --nologo letValueLF.fsx && mono letValueLF.exe
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 Listing 6.5.

· lightweight syntax

Listing 6.5 letValueLightWeight.fsx:
Lightweight syntax does not require the in keyword, but expression must be aligned with the let keyword.

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

```
$ fsharpc --nologo letValueLightWeight.fsx
$ mono letValueLightWeight.exe
9.0
```

The same expression in interactive mode will also respond the inferred types as, e.g., shown in Listing 6.6.

Listing 6.6: Interactive mode also responds inferred types.

```
> let p = 2.0
- do 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 as shown in Listing 6.7.

**Listing 6.7 letValueTypeError.fsx:
Binding error due to type mismatch.**

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

```
$ fsharpc --nologo letValueTypeError.fsx && mono letValueTypeError.exe

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 "`;`", see Listing 6.8.

**Listing 6.8 letValueSequence.fsx:
A value-binding for a sequence of expressions.**

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

```
$ fsharpc --nologo letValueSequence.fsx && mono letValueSequence.exe
2.0
9.0
```

The lightweight syntax automatically inserts the "`;`" lexeme at newlines, hence using the lightweight syntax the above is the same as shown in Listing 6.9.

**Listing 6.9 letValueSequenceLightWeight.fsx:
A value-binding for a sequence using lightweight syntax.**

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

```
$ fsharpc --nologo letValueSequenceLightWeight.fsx
$ mono letValueSequenceLightWeight.exe
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, see, e.g., Listing 6.10.

· scope

Listing 6.10 letValueScopeLower.fsx:
Redefining identifiers is allowed in lower scopes.

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

```
$ fsharpc --nologo letValueScopeLower.fsx && mono letValueScopeLower.exe
 4
```

Some special bindings are mutable, in which case F# uses the dynamic scope, that is, the value of a binding is defined by when it is used, and this will be discussed in Section 6.7.

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, as shown in Listing 6.11.

Listing 6.11 letValueScopeLowerError.fsx:
Redefining identifiers is not allowed in lightweight syntax at top level.

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

```
$ fsharpc --nologo -a letValueScopeLowerError.fsx

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, as demonstrated in Listing 6.12.

· block
· nested scope

Listing 6.12 letValueScopeBlockAlternative3.fsx:
A block may be created using parentheses.

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

```
$ fsharpc --nologo letValueScopeBlockAlternative3.fsx
$ mono letValueScopeBlockAlternative3.exe
4
```

In both cases, we used indentation, which is good practice, but not required here. Bindings inside are not available outside the nested scope as shown in Listing 6.13.

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

**Listing 6.13 letValueScopeNestedScope.fsx:
Bindings inside a scope are not available outside.**

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

```
$ fsharpc --nologo -a letValueScopeNestedScope.fsx

letValueScopeNestedScope.fsx(6,22): error FS0039: The value or
   constructor 'q' is not defined. Maybe you want one of the following:
   p
```

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 used for controlling the extent of a lexical scope of bindings. For example, adding a second `printfn` statement as in Listing 6.14, 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)`.

**Listing 6.14 letValueScopeBlockProblem.fsx:
Overshadowing hides the first binding.**

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

```
$ fsharpc --nologo letValueScopeBlockProblem.fsx
$ mono letValueScopeBlockProblem.exe
4
4
```

Had we intended to print the two different values of `p`, then we should have created a block as in Listing 6.15.

**Listing 6.15 letValueScopeBlock.fsx:
Blocks allow for the return to the previous scope.**

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

```
$ fsharpc --nologo letValueScopeBlock.fsx && mono letValueScopeBlock.exe
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 Function bindings

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,

Listing 6.16 Function binding expression

```
let <funcIdent> <arg> {<arg>} | () = <bodyExpr> [in <expr>]
```

Here `<funcIdent>` is an identifier and is the name of the function, `<arg>` is zero or more identifiers, that bind to the value used when calling the function, and which is to be used in the body of the function, the expression `<bodyExpr>`. The `|` notation denotes a choice, i.e., either that on the left-hand-side or that of the right-hand-side. Thus `let f x = x * x` and `let f () = 3` are valid function bindings, but `let f = 3` would be a value binding not a function binding. The arguments and the function may be annotated with a type, in which case for arguments we write

Listing 6.17 Function binding expression

```
let <funcIdent> (<arg> : <type>) {(<arg> : <type>)} : <type> | () : <type>
   = <bodyExpr> [in <expr>]
```

where `<type>` is a name of an existing type. The argument types are given In parentheses, and the return type is given last.

Functions are a key concept in F# and in this chapter, we will discuss the very basics. Recursive functions will be discussed in Chapter 8, and higher-order functions in Chapter 16.

An example of defining a function and using in interactive mode is shown in Listing 6.18.

Listing 6.18: 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
- do 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 as in Listing 6.19.

Listing 6.19 letFunctionAlterantive.fsx:
All types need most often not be specified.

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

Or even just one of the arguments as in Listing 6.20.

Listing 6.20 letFunctionAlterantive2.fsx:
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 "`;`" as shown in Listing 6.21.

· operator
· operand

Listing 6.21 letFunctionLightWeight.fsx:
Lightweight syntax for function definitions.

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

```
$ fsharpc --nologo letFunctionLightWeight.fsx
$ mono letFunctionLightWeight.exe
1221.0
```

Arguments need not always be inferred to types, but may be of the generic type, which F# prefers, when *type safety* is ensured as shown in Listing 6.22.

· type safety

Listing 6.22: Type safety implies that a function will work for any type, and hence it is generic.

```
> let second x y = y
- let a = second 3 5
- do printfn "%A" a
- let b = second "horse" 5.0
- do 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 shown in Listing 6.23.

Listing 6.23 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
do printfn "0 = %A * x ** 2.0 + %A * x + %A" a b c
do printfn "  has solutions %A and %A" xn xp
```

```
$ fsharpc --nologo identifiersExampleAdvance.fsx
$ mono identifiersExampleAdvance.exe
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 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 in Listing 6.24 shows.[2]

· lexical scope

Listing 6.24 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
do printfn "%A" (testScope 2.0)
```

```
$ fsharpc --nologo lexicalScopeNFunction.fsx
$ mono lexicalScopeNFunction.exe
6.0
```

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

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 * z` uses the value of `a`, as it is defined 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 * z`.

Advice

Functions do not need a name, but may be declared as an *anonymous function* using the `fun` keyword and the "`->`" lexeme, as shown in Listing 6.25.

· anonymous function
· `fun`
· `->`

Listing 6.25 functionDeclarationAnonymous.fsx:
Anonymous functions are functions as values.

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

```
$ fsharpc --nologo functionDeclarationAnonymous.fsx
$ mono functionDeclarationAnonymous.exe
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 6.7. A common use of anonymous functions is as arguments to other functions, as demonstrated in Listing 6.26.

Listing 6.26 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
do printfn "%d" (apply mul 3 6)
```

```
$ fsharpc --nologo functionDeclarationAnonymousAdvanced.fsx
$ mono functionDeclarationAnonymousAdvanced.exe
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 powerful concepts but tend to make programs harder to read, and their use should be limited.**

Advice

The result of one function is often used as an argument of another. This is function composition, and an example is shown in Listing 6.27.

**Listing 6.27 functionComposition.fsx:
Composing functions using intermediate bindings.**

```
let f x = x + 1
let g x = x * x

let a = f 2
let b = g a
let c = g (f 2)
do printfn "a = %A, b = %A, c = %A" a b c
```

```
$ fsharpc --nologo functionComposition.fsx
$ mono functionComposition.exe
a = 3, b = 9, c = 9
```

In the example we combine two functions `f` and `g` by storing the result of `f 2` in `a` and using that as argument of `g`. This is the same as `g (f 2)`, and in the later case, the compile creates a temporary value for `f 2`. Such compositions are so common in F# that a special operator has been invented called the *piping* operators, ">/" and "</". They are used as demonstrated in Listing 6.28.

· piping
· lstinline|>
· <|

**Listing 6.28 functionPiping.fsx:
Composing functions by piping.**

```
1 let f x = x + 1
2 let g x = x * x
3
4 let a = g (f 2)
5 let b = 2 |> f |> g
6 let c = g <| (f <| 2)
7 do printfn "a = %A, b = %A, c = %A" a b c
```

```
1 $ fsharpc --nologo functionPiping.fsx && mono functionPiping.exe
2 a = 9, b = 9, c = 9
```

The example shows regular composition and left-to-right and right-to-left piping. The word piping is a picture of data flowing through pipes, where functions are places, where the pipes have been assembled and the data changes. The three expressions in Listing 6.28 performes the same calculation. The left-to-right piping in line 5 corresponds to the left-to-right reading direction of many human languages, i.e., the value 2 is used as argument to `f`, and the result is used as argument to `g`. However, this is opposite arithmetic composition in line 4. Right-to-left piping has the order of arithmetic composition. Unfortunately, since F# evaluates the expression from the left, without the parenthesis in line 6, `g <| f <| 2`, then F# would read the expression as `(g <| f) <| 2`, which is an error, since `g` takes an integer as argument not a function. F# can also define composite on a function level, further discussion on this is deferred to Chapter 16. The piping operator comes in four variants: "|&gt;|", "&lt;||", "||&gt;|", and "&lt;|||". The allow for piping between pairs and triples to functions of 2 and 3 arguments, see Listing 6.29 for an example.

Listing 6.29 functionTuplePiping.fsx:
Tuples can be piped to functions of more than one argument.

```
let f x = printfn "%A" x
let g x y = printfn "%A %A" x y
let h x y z = printfn "%A %A %A" x y z

1 |> f
(1, 2) ||> g
(1, 2, 3) |||> h
```

```
$ fsharpc --nologo functionTuplePiping.fsx
$ mono functionTuplePiping.exe
1
1 2
1 2 3
```

The example demonstrates left-to-right piping, right-to-left works analogously.[3]

· procedure

A *procedure* is a generalization of the concept of functions, and in contrast to functions, procedures need not return values. This is demonstrated in Listing 6.30.

Listing 6.30 procedure.fsx:
A procedure is a function that has no return value, and in F# returns "()".

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

```
$ fsharpc --nologo procedure.fsx && mono procedure.exe
This is '3'
This is '3.0'
```

· encapsulation
· side-effects
Advice

In F# this is automatically given the unit type as the 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 6.7. **Procedural thinking is useful for encapsulation, but is prone to side-effects and should be minimized by being replaced by functional thinking.**

· first-class citizens
· closures

In F# functions (and procedures) are *first-class citizens*, which means that functions are values: They may be passed as arguments, returned from a function, and bound to a name. For first-class citizens, the name, it is bound to, does not carry significance to the language, as, e.g., illustrated with the use of anonymous functions. Technically, a function is stored as a *closures*. A closure is a description of the function, its arguments, its expression, and the environment, at the time it was created, i.e., the triple: $(args, exp, env)$. Consider the listing in Listing 6.31.

[3]Jon: **Tuples have not yet been introduced!**

Listing 6.31 functionFirstClass.fsx:
The function `timesTwo` has a non-trivial closure.

```
1 let mul x y = x * y
2 let factor = 2.0
3 let applyFactor fct x =
4   let a = fct factor x
5   string a
6
7 do printfn "%g" (mul 5.0 3.0)
8 do printfn "%s" (applyFactor mul 3.0)
```

```
1 $ fsharpc --nologo functionFirstClass.fsx && mono functionFirstClass.exe
2 15
3 6
```

It defines two functions `mul` and `applyFactor`, where the later is a higher order function taking another function as an argument and uses part of the environment to produce its result. The two closures are,

$$\texttt{mul} : (\text{args}, \text{exp}, \text{env}) = ((\text{x}, \text{y}), (\texttt{x * y}), ()) \tag{6.3}$$

$$\texttt{applyFactor} : (\text{args}, \text{exp}, \text{env}) = \left((\text{x}, \texttt{fct}), \left(\begin{matrix}\texttt{let a = fct factor x}\\ \texttt{string a}\end{matrix}\right), (\texttt{factor} \rightarrow 2.0)\right) \tag{6.4}$$

The function `mul` does not use its environment, and everything needed to evaluate its expression are values for its arguments. The function `applyFactor` also takes two arguments, a function and a value. It uses `factor` from the environment, thus this is stored in its closure. When `mul` is given as an argument in Listing 6.31 line 8, then it is its closure, which is given to `applyFactor`, and the closure contains everything that `applyFactor` requires to use `mul`. Likewise, if `applyFactor` is given as argument to yet another function, then its closure includes the relevant part of its environment at time of definition, i.e., `factor`, such that when `applyFactor` is applied to two arguments, then its closure contains everything needed to evaluate its expression.

6.3 Operators

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

Listing 6.32 addOperatorNFunction.fsx:
Operators have function equivalents.

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

```
$ fsharpc --nologo addOperatorNFunction.fsx
$ mono addOperatorNFunction.exe
3.0 plus 4.0 is 7.0 and 7.0
```

All operators have this option, and you may redefine them and define your own operators, but in F#

names of user-defined operators are limited.

· operator

operator

- A *unary operator* name can be: “+”, “-”, “+.”, “-.”, “%”, “&”, “&&”, “~~”, “~~~”, “~~~~”, . . . , apostropheOp. Here apostropheOp is an operator name starting with “!” followed by one or more of either “!”, “%”, “&”, “*”, “+”, “-”, “.”, “/”, “<”, “=”, “>”, “@”, “^”, “|”, “~”, but apostropheOp cannot be “!=”. · unary operator
- An *binary operator* name can be: “+”, “-”, “+.”, “-.”, “%”, “&”, “&&”, “:=”, “::”, “$”, “?”, dotOp. Here dotOp is an operator name starting with “.” followed by “+”, “-”, “+.”, “-.”, “%”, “&”, “&&”, “-”, “+”, “||”, “<”, “>”, “=”, “|”, “&”, “^”, “*”, “/”, “%”, “!=”. Only “?” and “?<-” may start with “?”. · binary operator

The precedence rules and associativity of user-defined operators follow the rules for which they share prefixes with built-in rules, see Table E.5. E.g., .*, +++, and <+ are valid operator names for infix operators, they have precedence as ordered, and their associativities 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 6.33 operatorDefinitions.fsx:
Operators may be (re)defined by their function equivalent.

```
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)
```

```
$ fsharpc --nologo operatorDefinitions.fsx
$ mono operatorDefinitions.exe
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 the 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 Do bindings

Aside from let bindings that binds names with values or functions, sometimes we just need to execute code. This is called a *do* binding or alternatively a *statement*. The syntax is as follows.

· do
· do binding
· statement

Listing 6.34 Syntax for do bindings.

```
[do ]<expr>
```

The expression `<expr>` must return `unit`. The keyword `do` is optional in most cases, but using it emphasizes that the expression is not a function that returns a useful value. Procedures are examples of such expressions and a very useful family of procedures are the `printf` family described below. In the remainder of this book, we will refrain from using the `do` keyword.

6.5 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 6.35 printf statement.

```
printf <format-string> {<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`. There are specifiers for all the basic types and more as elaborated in Table 6.2. 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. E.g., the placeholder string `%8s` will print a right-aligned string that is eight characters wide padded with spaces, if needed. For floating point numbers, `%8f` will print a number that is exactly seven digits and a decimal making eight characters in total. Zeros are added after the decimal as needed. Alternatively, we may specify the number of decimals, such that `%8.1f` will print a floating point number, aligned to the right with one digit after the decimal padded with spaces where needed. The 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 right justifying the number; `"+"` to explicitly show a plus sign for positive numbers; `SP` to enforce a space, where there otherwise would be a plus sign for positive numbers. Examples of some of these combinations are shown in Listing 6.36.

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 6.2: Printf placeholder string

Listing 6.36 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\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"
```

```
$ fsharpc --nologo printfExample.fsx && mono printfExample.exe
An integer: 3
A float 3.141519 on decimal form and on 3.141519e+000 scientific form
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 6.3: 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 shown in Listing 6.37.

Listing 6.37 printfExampleAdvance.fsx:
Custom format functions may be used to specialise output.

```
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
```

```
$ fsharpc --nologo printfExampleAdvance.fsx
$ mono printfExampleAdvance.exe
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 11 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 6.3. 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 19. The function `failwithf` is used with exceptions, see Chapter 18 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`

6.6 Reading from the console

The `printf` and `printfn` functions allow us to write text on the screen, i.e., `stdout`. A program often needs to ask a user to input data, e.g., by typing text on a keyboard. Text typed on the keyboard is accessible through the `stdin` stream, and F# provides several library functions for capturing text typed on the keyboard. In the following, we will briefly discuss the `System.Console.ReadLine` function. For more details and other methods of input see Chapters 19 and 23.

The function `System.Console.ReadLine` takes a unit value as an argument and returns the string the user typed. The program will not advance until the user presses newline. An example of a program that multiplies two floating point numbers supplied by a user is given in Listing 6.38,

**Listing 6.38 userDialoguePrintf.fsx:
Interacting with a user with `ReadLine`.**

```
printfn "To perform the multiplication of a and b"
printf "Enter a: "
let a = float (System.Console.ReadLine ())
printf "Enter b: "
let b = float (System.Console.ReadLine ())
printfn "a * b = %A" (a * b)
```

and an example dialogue is shown in Listing 6.39.

Listing 6.39: An example dialogue of running Listing 6.38. What the user typed has been framed in red boxes.

```
$ fsharpc --nologo userDialoguePrintf.fsx && mono userDialoguePrintf.exe
To perform the multiplication of a and b
Enter a: 3.5
Enter b: 7.4
a * b = 25.9
```

Note that the string is immediately cast to floats, such that we can multiply the input using the float multiplication operator. This also implies, that if the user inputs anything but floating point value, then `mono` will halt with an error message.

6.7 Variables

The *`mutable`* in `let` bindings means that the identifier may be rebound to a new value using the "`<-`" lexeme with the following syntax.[4]

· `mutable`
· `<-`

Listing 6.40 Syntax for defining variables and their initial value.

```
let mutable <ident> = <expr> [in <expr>]
```

and values are changed using the assignment operator,

[4]Jon: **Discussion on heap and stack should be added here.**

Listing 6.41 Value reassignment for mutable variables.

```
<ident> <- <ident>
```

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, see Listing 6.42 for an example.

· mutable data
· variable

Listing 6.42 mutableAssignReassingShort.fsx:
A variable is defined and later reassigned a new value.

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

```
$ fsharpc --nologo mutableAssignReassingShort.fsx
$ mono mutableAssignReassingShort.exe
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 as shown in Listing 6.43, 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.

Listing 6.43: It is a common error to mistake "=" and "<-" lexemes for mutable variables.

```
> let mutable a = 0
- a = 3;;
val mutable a : int = 0
val it : bool = false
```

However, it is 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 as demonstrated in Listing 6.44.

**Listing 6.44 mutableAssignReassingTypeError.fsx:
Assignment type mismatching causes a compile-time error.**

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

```
$ fsharpc --nologo mutableAssignReassingTypeError.fsx

mutableAssignReassingTypeError.fsx(3,6): error FS0001: This expression
   was expected to have type
    'int'
but here has type
    'float'
$ mono mutableAssignReassingTypeError.exe
Cannot open assembly 'mutableAssignReassingTypeError.exe': No such file
   or directory.
```

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, see Listing 6.45 for an example.

**Listing 6.45 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
```

```
$ fsharpc --nologo mutableAssignIncrement.fsx
$ mono mutableAssignIncrement.exe
5
6
```

Using variables in expressions, as opposed to the left-hand-side of an assignment operation, reads the value of the variable. Thus, when using a variable as the return value of a function, then the value is copied from the local scope of the function to the scope from which it is called. This is demonstrated in Listing 6.46.

Listing 6.46: Returning a mutable variable returns its value.

```
> let g () =
-   let mutable y = 0
-   y
- printfn "%d" (g ());;
0
val g : unit -> int
val it : unit = ()
```

In the example, we see that the type is a value, and not mutable.

Variables implement dynamic scope, e.g., in comparison with the lexical scope, where the value of an identifier depends on *where* it is defined, dynamic scope depends on, *when* it is used. E.g., the script in Listing 6.24 defines a function using lexical scope and returns the number `6.0`, however, if `a` is made `mutable`, then the behavior is different as shown in Listing 6.47.

Listing 6.47 dynamicScopeNFunction.fsx:
Mutual variables implement dynamics scope rules. Compare with Listing 6.24.

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

```
$ fsharpc --nologo dynamicScopeNFunction.fsx
$ mono dynamicScopeNFunction.exe
8.0
```

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

6.8 Reference cells

F# has a variation of mutable variables called *reference cells*. Reference cells have built-in function *`ref`* and operators "`!`" and "`:=`", where `ref` creates a reference variable, and the "'`!`'" and the "'`:=`'" operators reads and writes its value. An example of using reference cells is given in Listing 6.48.

· reference cells
· `ref`
· `:=`

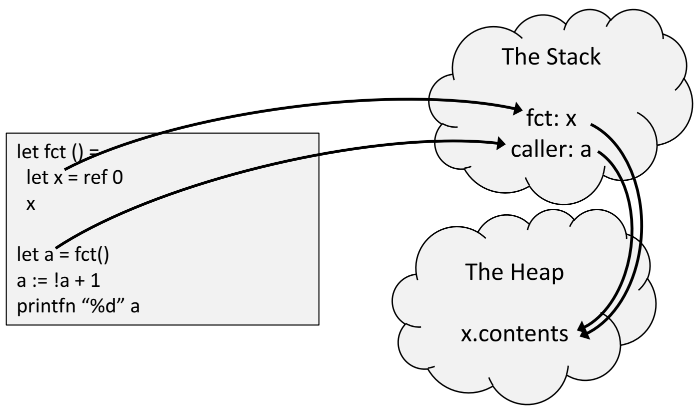
Listing 6.48 refCell.fsx:
Reference cells are variants of mutable variables.

```
let x = ref 0
printfn "%d" !x
x := !x + 1
printfn "%d" !x
```

```
$ fsharpc --nologo refCell.fsx && mono refCell.exe
0
1
```

Reference cells are different from mutable variables since their content is allocated on *The Heap*. The Heap is a global data storage that is not destroyed, when a function returns, which is in contrast to the *call stack* also known as *The Stack*. The Stack maintains all the local data for a specific instance of a function call, see Section 13.2 for more detail on the call stack. As a consequence, when a reference cell is returned from a function, then it is the reference to the location on The Heap, which is returned as a value. Since this points outside the local data area of the function, then this location is still valid after the function returns, and the variable stored there is accessible to the caller. This is illustrated in Figure 6.1

· The Heap
· call stack
· The Stack


```
let fct () =
  let x = ref 0
  x

let a = fct()
a := !a + 1
printfn "%d" a
```

Figure 6.1: A reference cell is a pointer to The Heap and the content is not destroyed when its reference falls out of scope.

Reference cells may cause *side-effects*, where variable changes are performed across independent scopes. · side-effects
Some side-effects are useful, e.g., the `printf` family changes the content of the screen, and the screen is outside the scope of the caller. Another example of a useful side-effect is a counter shown in Listing 6.49.

Listing 6.49 refEncapsulation.fsx:
An increment function with a local state using a reference cell.

```
let incr =
  let counter = ref 0
  fun () ->
    counter := !counter + 1
    !counter
printfn "%d" (incr ())
printfn "%d" (incr ())
printfn "%d" (incr ())
```

```
$ fsharpc --nologo refEncapsulation.fsx && mono refEncapsulation.exe
1
2
3
```

Here `incr` is an anonymous function with an internal state `counter`. At first glance, it may be surprising, that `incr ()` does not return the value `1` at every call. The reason is that the value of the `incr` is the closure of the anonymous function `fun () -> counter := ...`, which is

$$\texttt{incr} : (\text{args}, \text{exp}, \text{env}) = \left((), \left(\begin{array}{l}\texttt{counter := !counter + 1}\\ \texttt{!counter}\end{array}\right), (\texttt{counter} \to \texttt{ref 0})\right). \tag{6.5}$$

Thus, `counter` is only initiated once at the initial binding, while every call of `incr ()` updates its value on The Heap. Such a programming structure is called *encapsulation*, since the `counter` state · encapsulation
has been encapsulated in the anonymous function, and the only way to access it is by calling the same anonymous function. **Encapsulation is good programming practice, but side-effects should be avoided wherever possible.** Advice

The `incr` example in Listing 6.49 is an example of a useful side-effect. An example to be avoided is shown in Listing 6.50.

Listing 6.50 refSideEffect.fsx:
Intertwining independent scopes is typically a bad idea.

```
let updateFactor factor =
  factor := 2

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

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

```
$ fsharpc --nologo refSideEffect.fsx && mono refSideEffect.exe
6
```

In the example, the function `updateFactor` changes a variable in the scope of `multiplyWithFactor`, which is prone to errors, since the computations are not local at the place of writing, i.e., in `multiplyWithFactor`, and if `updateFactor` were defined in a library, then the source code may not be available. Better style of programming is shown in Listing 6.51.

Listing 6.51 refWithoutSideEffect.fsx:
A solution of Listing 6.50 avoiding side-effects.

```
let updateFactor () =
  2

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

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

```
$ fsharpc --nologo refWithoutSideEffect.fsx
$ mono refWithoutSideEffect.exe
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 **minimize the use of side effects**. Advice

Reference cells give rise to an effect called *aliasing*, where two or more identifiers refer to the same data as illustrated in Listing 6.52. · aliasing

Listing 6.52 refCellAliasing.fsx:
Aliasing can cause surprising results and should be avoided.

```
let a = ref 1
let b = a
printfn "%d, %d" !a !b
b := 2
printfn "%d, %d" !a !b
```

```
$ fsharpc --nologo refCellAliasing.fsx && mono refCellAliasing.exe
1, 1
2, 2
```

Here, `a` is defined as a reference cell, and by defining `b` to be equal to `a`, we have created an alias. This can be very confusing since as the example shows, changing the value of `b` causes `a` to change as well. Aliasing is a variant of side-effects, and **aliasing should be avoided at all costs**. Advice

Since F# version 4.0, the compiler has automatically converted mutable variables to reference cells, when needed. E.g., Listing 6.49 can be rewritten using a mutable variable as shown in Listing 6.53.

Listing 6.53 mutableEncapsulation.fsx:
Local mutable content can be indirectly accessed outside its scope.

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

```
$ fsharpc --nologo mutableEncapsulation.fsx
$ mono mutableEncapsulation.exe
1
2
3
```

Reference cells are preferred over mutable variables for encapsulation to avoid confusion.

6.9 Tuples

Tuples are a direct extension of constants. They are immutable and do not have concatenations nor indexing operations. Tuples are unions of immutable types and have the following syntax, · tuple

Listing 6.54

```
<expr>{, <expr>}
```

Tuples are identified by the "`,`" lexeme and often enclosed in parentheses, but that is not required. An example is a triple, also known as a 3-tuple, `(2,true,"hello")`. In interactive mode, the type of tuples is shown as demonstrated in Listing 6.55.

Listing 6.55: Tuple types are products of sets.

```
> let tp = (2, true, "hello")
- printfn "%A" tp;;
(2, true, "hello")
val tp : int * bool * string = (2, true, "hello")
val it : unit = ()
```

The values 2, `true`, and `"hello"` are *members*, and the number of elements of a tuple is its *length*. From the response of F#, we see that the tuple is inferred to have the type `int * bool * string`. The "`*`" denotes the Cartesian product between sets. Tuples can be products of any types and have the lexical scope like value and function bindings. Notice also that a tuple may be printed as a single entity by the `%A` placeholder. In the example, we bound `tp` to the tuple, the opposite is also possible, where an identifier is bound to a tuple as shown in Listing 6.56.

· member
· length

Listing 6.56: Definition of a tuple.

```
> let deconstructNPrint tp =
-   let (a, b, c) = tp
-   printfn "tp = (%A, %A, %A)" a b c
-
- deconstructNPrint (2, true, "hello")
- deconstructNPrint (3.14, "Pi", 'p');;
tp = (2, true, "hello")
tp = (3.14, "Pi", 'p')
val deconstructNPrint : 'a * 'b * 'c -> unit
val it : unit = ()
```

In this, a function is defined that takes 1 argument, a 3-tuple. If we wanted a function of 3 arguments, then the function binding should have been `let deconstructNPrint a b c = ...`. The value binding `let (a, b, c) = tp`, binds a tuple with 3 named members to a value, thus deconstructing it in terms of its members. This is called pattern matching and will be discussed in further details in Chapter 15. Since we used the `\%A` placeholder in the `printfn` function, then the function can be called with 3-tuples of different types. F# informs us that the tuple type is variable by writing `'a * 'b * 'c`. The "`'`" notation means that the type can be decided at run-time, see Section 14.6 for more on variable types.

Pairs or 2-tuples are so common that F# includes two built-in functions *`fst`* and *`snd`*, to extract the first and second element of a pair. This is demonstrated in Listing 6.57.

· `fst`
· `snd`

**Listing 6.57 pair.fsx:
Deconstruction of pairs with the built-in functions `fst` and `snd`.**

```
let pair = ("first", "second")
printfn "fst(pair) = %s, snd(pair) = %s" (fst pair) (snd pair)
```

```
$ fsharpc --nologo pair.fsx && mono pair.exe
fst(pair) = first, snd(pair) = second
```

Tuples comparison are defined similarly as strings. Tuples of different lengths are different. For tuples of equal length, then they are compared element by element. E.g., `(1,2) = (1,3)` is false, while `(1,2) = (1,2)` is true. The "<>" operator is the boolean negation of the "=" operator. For the "<", "<=", ">", and ">=" operators, the strings are ordered lexicographically, such that `('a', 'b', 'c') < ('a', 'b', 's') && ('a', 'b', 's') <  ('c', 'o', 's')` is true, that is, the "<" operator on two tuples is true, if the left operand should come before the right, when sorting alphabetically, see Listing 6.58 for an example.

Listing 6.58 tupleCompare.fsx:
Tuples are compared as strings are compared alphabetically.

```fsharp
let lessThan (a, b, c) (d, e, f) =
  if a <> d then a < d
  elif b <> e then b < d
  elif c <> f then c < f
  else false

let printTest x y =
  printfn "%A < %A is %b" x y (lessThan x y)

let a = ('a', 'b', 'c');
let b = ('d', 'e', 'f');
let c = ('a', 'b', 'b');
let d = ('a', 'b', 'd');
printTest a b
printTest a c
printTest a d
```

```
$ fsharpc --nologo tupleCompare.fsx && mono tupleCompare.exe
('a', 'b', 'c') < ('d', 'e', 'f') is true
('a', 'b', 'c') < ('a', 'b', 'b') is false
('a', 'b', 'c') < ('a', 'b', 'd') is true
```

The algorithm for deciding the boolean value of `(a1, a2) < (b1, b2)` is as follows: we start by examining the first elements, and if `a1` and `b1` are different, in which case the result of `(a1, a2) < (b1, b2)` is equal to the result of `a1 < b1`. If `la1` and `b1` are equal, then we move onto the next letter and repeat the investigation. The "<=", ">", and ">=" operators are defined similarly.

Binding tuples to mutables does not make the tuple mutable. This is demonstrated in Listing 6.59.

Listing 6.59 tupleOfMutables.fsx:
A mutable change value, but the tuple defined by it does not refer to the new value.

```fsharp
let mutable a = 1
let mutable b = 2
let c = (a, b)
printfn "%A, %A, %A" a b c
a <- 3
printfn "%A, %A, %A" a b c
```

```
$ fsharpc --nologo tupleOfMutables.fsx && mono tupleOfMutables.exe
1, 2, (1, 2)
3, 2, (1, 2)
```

However, it is possible to define a mutable variable of type tuple such that new tuple values can be assigned to it as shown in Listing 6.60.

Listing 6.60 mutableTuple.fsx:
A mutable tuple can be assigned a new value.

```
let mutable pair = 1,2
printfn "%A" pair
pair <- (3,4)
printfn "%A" pair
```

```
$ fsharpc --nologo mutableTuple.fsx && mono mutableTuple.exe
(1, 2)
(3, 4)
```

Mutable tuples are value types meaning that binding to new names makes copies not aliases, as demonstrated in Listing 6.61.

Listing 6.61 mutableTupleValue.fsx:
A mutable tuple is a value type.

```
let mutable pair = 1,2
let mutable aCopy = pair
pair <- (3,4)
printfn "%A %A" pair aCopy
```

```
$ fsharpc --nologo mutableTupleValue.fsx && mono mutableTupleValue.exe
(3, 4) (1, 2)
```

The use of tuples shortens code and highlights semantic content at a higher level, e.g., instead of focussing on the elements, tuples focus on their union. While this may look elegant and short there is the risk of *obfuscation*, i.e., writing compact code that is difficult to read, where an unprepared reader of the code may not easily understand the computation nor appreciate its elegance without an accompanying explanation. Hence, **always keep an eye out for compact and concise ways to write code, but never at the expense of readability.**

· obfuscation

Advice

7 In-code documentation

Documentation is a very important part of writing programs, since it is most unlikely, that you will be writing really obvious code. Moreover, what seems obvious at the point of writing may be mystifying months later to the author and to others. Documentation serves several purposes:

1. Communicate what the code should be doing
2. Highlight big insights essential for the code
3. Highlight possible conflicts and/or areas, where the code could be changed later

The essential point is that coding is a journey in problem-solving, and proper documentation is an aid in understanding the solution and the journey that lead to it. Documentation is most often a mixture of in-code documentation and accompanying documents. Here, we will focus on in-code documentation, which arguably causes problems in multi-language environments and run the risk of bloating code.

F# has two different syntaxes for comments. Comments are either block comments,

Listing 7.1 Block comments.

```
(<any text>)
```

The comment text `(<any text>)` can be any text and it is stilled parsed by F# as keywords and basic types implying that `(* a comment (* in a comment *) *)` and `(* "*)" *)` are valid comments, while `(* " *)` is invalid.

Alternatively, comments may be given as line comments,

Listing 7.2 Line comments.

```
//<any text>newline
```

where the comment text ends after the first newline.

The F# compiler has an option for generating *Extensible Markup Language* (*XML*) files from scripts using the C# documentation comments tags[1]. The XML documentation starts with a triple-slash `///`, i.e., a lineComment and a slash, which serves as comments for the code construct, that follows immediately after. XML consists of tags which always appears in pairs, e.g., the tag "tag" would look like `<tag> ... </tag>`. The F# accept any tags, but recommends those listed in Table 7.1. If no tags

· Extensible Markup Language
· XML

[1]For specification of C# documentations comments see ECMA-334 3rd Edition, Annex E, Section 2: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf

Tag	Description
`<c>`	Set text in a code-font.
`<code>`	Set one or more lines in code-font.
`<example>`	Set as an example.
`<exception>`	Describe the exceptions a function can throw.
`<list>`	Create a list or table.
`<para>`	Set text as a paragraph.
`<param>`	Describe a parameter for a function or constructor.
`<paramref>`	Identify that a word is a parameter name.
`<permission>`	Document the accessibility of a member.
`<remarks>`	Further describe a function.
`<returns>`	Describe the return value of a function.
`<see>`	Set as link to other functions.
`<seealso>`	Generate a See Also entry.
`<summary>`	Main description of a function or value.
`<typeparam>`	Describe a type parameter for a generic type or method.
`<typeparamref>`	Identify that a word is a type parameter name.
`<value>`	Describe a value.

Table 7.1: Recommended XML tags for documentation comments, from ECMA-334 3rd Edition, Annex E, Section 2.

are used, then it is automatically assumed to be a `<summary>`. An example of a documented script is,

**Listing 7.3 commentExample.fsx:
Code with XML comments.**

```
/// The discriminant of a quadratic equation with parameters a, b, and c
let discriminant a b c = b ** 2.0 - 4.0 * a * c

/// <summary>Find x when 0 = ax^2+bx+c.</summary>
/// <remarks>Negative discriminant are not checked.</remarks>
/// <example>
///   The following code:
///   <code>
///     let a = 1.0
///     let b = 0.0
///     let c = -1.0
///     let xp = (solution a b c +1.0)
///     printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
///   </code>
///   prints <c>0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7</c> to the console.
/// </example>
/// <param name="a">Quadratic coefficient.</param>
/// <param name="b">Linear coefficient.</param>
/// <param name="c">Constant coefficient.</param>
/// <param name="sgn">+1 or -1 determines the solution.</param>
/// <returns>The solution to x.</returns>
let solution a b c sgn =
  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)
printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
```

```
$ fsharpc --nologo commentExample.fsx && mono commentExample.exe
0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 1.0
```

Mono's **fsharpc** command may be used to extract the comments into an XML file as demonstrated in Listing 7.4.

Listing 7.4, Converting in-code comments to XML.

```
$ fsharpc --doc:commentExample.xml commentExample.fsx
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
```

This results in an XML file with the following content as shown in Listing 7.5.

Listing 7.5, An XML file generated by `fsharpc`.

```
<?xml version="1.0" encoding="utf-8"?>
<doc>
<assembly><name>commentExample</name></assembly>
<members>
<member name="M:CommentExample.solution(System.Double,System.Double,
   System.Double,System.Double)">
 <summary>Find x when 0 = ax^2+bx+c.</summary>
 <remarks>Negative discriminant are not checked.</remarks>
 <example>
   The following code:
   <code>
     let a = 1.0
     let b = 0.0
     let c = -1.0
     let xp = (solution a b c +1.0)
     printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
   </code>
   prints <c>0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7</c> to the console.
 </example>
 <param name="a">Quadratic coefficient.</param>
 <param name="b">Linear coefficient.</param>
 <param name="c">Constant coefficient.</param>
 <param name="sgn">+1 or -1 determines the solution.</param>
 <returns>The solution to x.</returns>
</member>
<member name="M:CommentExample.discriminant(System.Double,System.Double,
   System.Double)">
<summary>
 The discriminant of a quadratic equation with parameters a, b, and c
</summary>
</member>
</members>
</doc>
```

The extracted XML is written in C# type by convention, since F# is part of the Mono and .Net framework that may be used by any of the languages using Assemblies. Besides the XML inserted in the script, the XML has added `<?xml ...>` header, `<doc>`, `<assembly>`, `<members>`, and `<member>` tags. The header and the `<doc>` tag are standards for XML. The extracted XML is geared towards documenting big libraries of codes and thus highlights the structured programming organisation, see Chapters 9 and 20, and `<assembly>`, `<members>`, and `<member>` are indications for where the functions belong in the hierarchy. As an example, the prefix `M:CommentExample.` means that it is a method in the namespace `commentExample`, which in this case is the name of the file. Further, the function type `val solution : a:float -> b:float -> c:float -> sgn:float -> float` is in the XML documentation `M:CommentExample.solution(System.Double,System.Double,System.Double,System.Double)`, which is the C# equivalent.

An accompanying program in the Mono suite is `mdoc`, whose primary use is to perform a syntax analysis of an assembly and generate a scaffold XML structure for an accompanying document. With the `-i` flag, it is further possible to include the in-code comments as initial descriptions in the XML. The XML may be updated gracefully by `mdoc` as the code develops, without destroying manually entered documentation in the accompanying documentation. Finally, the XML may be exported to HTML.

The primary use of the `mdoc` command is to analyze compiled code and generate an empty XML structure with placeholders to describe functions, values, and variables. This structure can then be updated and edited as the program develops. The edited XML files can then be exported to *Hyper*

Text Markup Language (*HTML*) files, which can be viewed in any browser. Using the console, all of this is accomplished by the procedure shown in Listing 7.6, and the result is shown in Figure 7.1.

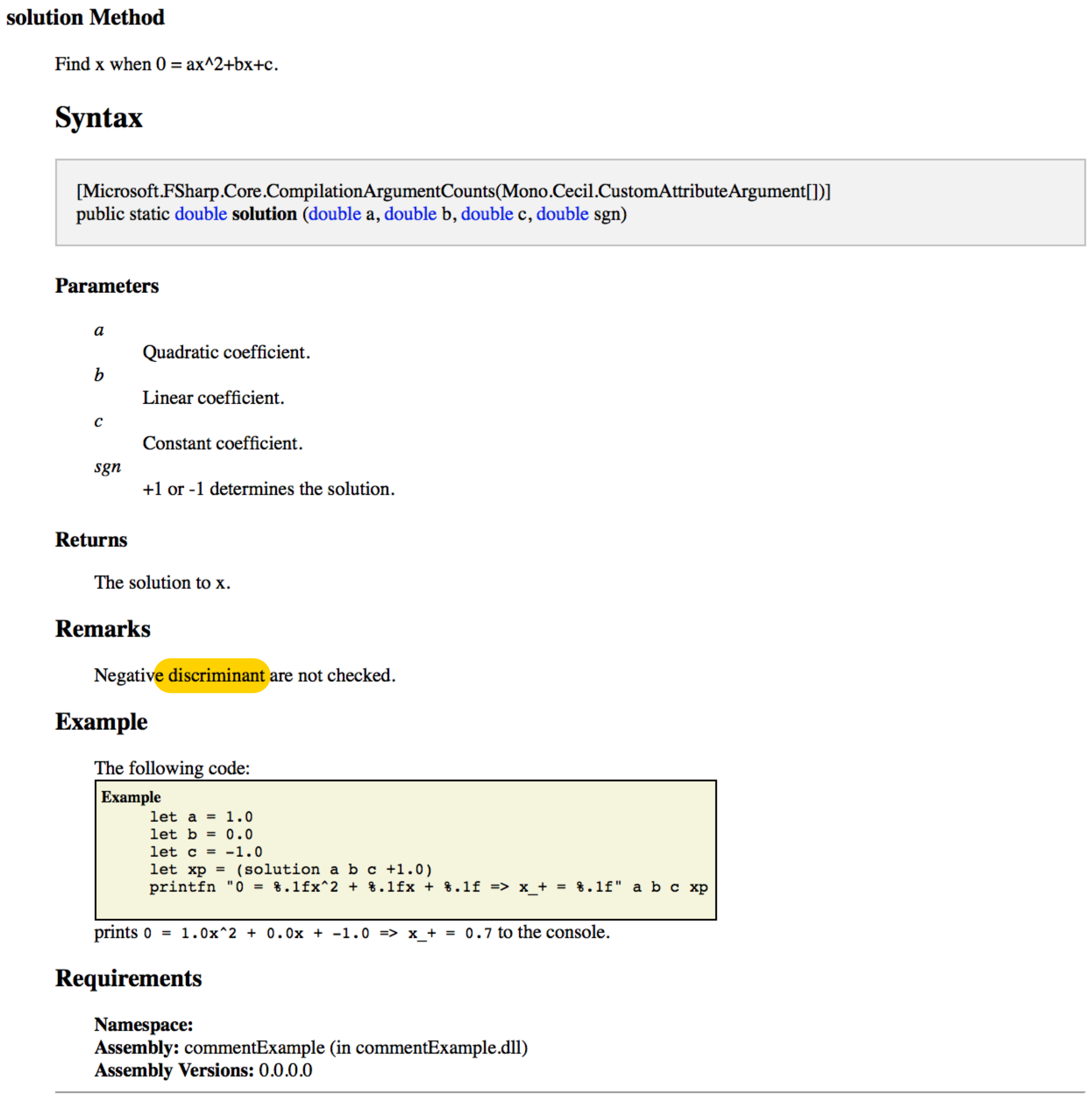
· Hyper Text Markup Language
· HTML

Listing 7.6, Converting an XML file to HTML.

```
$ mdoc update -o commentExample -i commentExample.xml commentExample.exe
New Type: CommentExample
Member Added: public static double determinant (double a, double b,
   double c);
Member Added: public static double solution (double a, double b, double
   c, double sgn);
Member Added: public static double a { get; }
Member Added: public static double b { get; }
Member Added: public static double c { get; }
Member Added: public static double xp { get; }
Namespace Directory Created:
New Namespace File:
Members Added: 6, Members Deleted: 0
$ mdoc export-html -out commentExampleHTML commentExample
.CommentExample
```

A full description of how to use `mdoc` is found here[2].

[2] http://www.mono-project.com/docs/tools+libraries/tools/monodoc/generating-documentation/



solution Method

Find x when 0 = ax^2+bx+c.

Syntax

```
[Microsoft.FSharp.Core.CompilationArgumentCounts(Mono.Cecil.CustomAttributeArgument[])]
public static double solution (double a, double b, double c, double sgn)
```

Parameters

a
Quadratic coefficient.
b
Linear coefficient.
c
Constant coefficient.
sgn
+1 or -1 determines the solution.

Returns

The solution to x.

Remarks

Negative discriminant are not checked.

Example

The following code:

Example
```
    let a = 1.0
    let b = 0.0
    let c = -1.0
    let xp = (solution a b c +1.0)
    printfn "0 = %.1fx^2 + %.1fx + %.1f => x_+ = %.1f" a b c xp
```

prints `0 = 1.0x^2 + 0.0x + -1.0 => x_+ = 0.7` to the console.

Requirements

Namespace:
Assembly: commentExample (in commentExample.dll)
Assembly Versions: 0.0.0.0

Figure 7.1: Part of the HTML documentation as produce by `mdoc` and viewed in a browser.

8 Controlling program flow

Non-recursive functions encapsulate code and allow for control of execution flow. That is, if a piece of code needs to be executed many times, then we can encapsulate it in the body of a function, and call the function several times. In this chapter, we will look at more general control of flow via loops and conditional execution. Recursion is another mechanism for controlling flow, but this is deferred to Chapter 13.

8.1 While and for loops

Many programming constructs need to be repeated, and F# contains many structures for repetition. A *while*-loop has the syntax,

· while

Listing 8.1 While loop.

```
while <condition> do <expr> [done]
```

The *condition* `<condition>` is an expression that evaluates to true or false. A while-loop repeats the `<expr>` expression as long as the condition is true. Using lightweight syntax the block following the *do* keyword up to and including the *done* keyword may be replaced by a newline and indentation. As an example, the program in Listing 8.5 counts from 1 to 10.

· condition
· do
· done

**Listing 8.2 countWhile.fsx:
Count to 10 with a counter variable.**

```
1 let mutable i = 1 in while i <= 10 do printf "%d " i; i <- i + 1 done;
2 printf "\n"
```

```
1 $ fsharpc --nologo countWhile.fsx && mono countWhile.exe
2 1 2 3 4 5 6 7 8 9 10
```

We will call i for the counter variable. The counting is done by performing the following computation: In line 1, the counter variable is first given an initial value of 1. Then execution enters the while-loop and examines the condition. Since 1 <= 10 then the condition is true, and execution enters the body of the loop. The body prints the value of the counter to the screen and increases counter by 1. Then execution returns to the top of the while-loop. Now the condition is 2 <= 10 which is also true, and so execution enters the body and so on until the counter has reach value 11, in which case the condition 11 <= 10 is false, and execution continues in line 2.

In lightweight syntax, this would be as shown in Listing 8.3.

Listing 8.3 countWhileLightweight.fsx:
Count to 10 with a counter variable using lightweight syntax, see Listing 8.2.

```
let mutable i = 1
while i <= 10 do
  printf "%d " i
  i <- i + 1
printf "\n"
```

```
$ fsharpc --nologo countWhileLightweight.fsx
$ mono countWhileLightweight.exe
1 2 3 4 5 6 7 8 9 10
```

Notice that although the expression following the condition is preceded with a `do` keyword, and `do <expr>` is a do binding, the keyword `do` is mandatory.

Counters are so common that a special syntax has been reserved for loops using counters. These are called *`for`*-loops. For-loops comes in several variants, and here we will focus on the one using an explicit counter. Its syntax is, · `for`

Listing 8.4 For loop.

```
for <ident> = <firstExpr> to <lastExpr> do <bodyExpr> [done]
```

A for-loop initially binds the counter identifier `<ident>` to be the value `<firstExpr>`. Then execution enters the body and `<bodyExpr>` is evaluated. Once done, then the counter is increased and execution evaluates once again `<bodyExpr>`. This is repeated until and including the counter has the value `<lastExpr>`. As for while-loops, using lightweight syntax the block following the *`do`* keyword up to and including the *`done`* keyword may be replaced by a newline and indentation. · `do` · `done`

The counting example from Listing 8.2 using a `for`-loop is shown in Listing 8.5

Listing 8.5 count.fsx:
Counting from 1 to 10 using a `for`-loop.

```
for i = 1 to 10 do printf "%d " i done
printfn ""
```

```
$ fsharpc --nologo count.fsx && mono count.exe
1 2 3 4 5 6 7 8 9 10
```

As this interactive script demonstrates, the identifier `i` takes all the values between 1 and 10, but in spite of its changing state, it is not mutable. Note also that the return value of the `for` expression is "`()`" like the `printf` functions. The lightweight equivalent is shown in Listing 8.6.

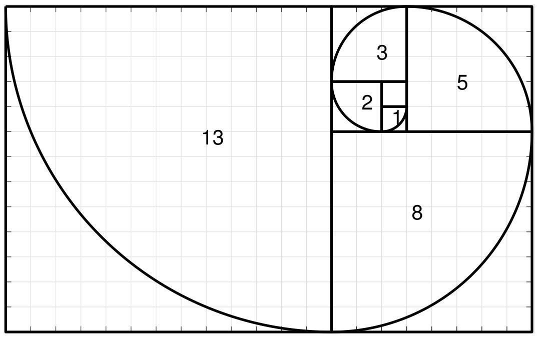


Figure 8.1: The Fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive Fibonacci numbers, and the curve in each square is the circular arc with a radius of the square it is drawn in.

**Listing 8.6 countLightweight.fsx:
Counting from 1 to 10 using a for-loop, see Listing 8.5.**

```
for i = 1 to 10 do
  printf "%d " i
printfn ""
```

```
$ fsharpc --nologo countLightweight.fsx && mono countLightweight.exe
1 2 3 4 5 6 7 8 9 10
```

To further compare for- and while-loops, consider the following problem.

Problem 8.1

Write a program that calculates the n'th Fibonacci number.

The Fibonacci's numbers is a sequence of numbers starting with $1, 1$, and where the next number is calculated as the sum of the previous two. Hence the first ten numbers are: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55$. Fibonacci's numbers are related to Golden spirals shown in Figure 8.1. Often the sequence is extended with a preceding number 0, to be $0, 1, 1, 2, 3, \ldots$, which we will do here as well.

We could solve this problem with a `for`-loop as follows,

Listing 8.7 fibFor.fsx:
The n'th Fibonacci number is the sum of the previous 2.

```
let fib n =
  let mutable pair = (0, 1)
  for i = 2 to n do
    pair <- (snd pair, (fst pair) + (snd pair))
  snd pair

printfn "fib(1) = %d" (fib 1)
printfn "fib(2) = %d" (fib 2)
printfn "fib(3) = %d" (fib 3)
printfn "fib(10) = %d" (fib 10)
```

```
$ fsharpc --nologo fibFor.fsx && mono fibFor.exe
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(10) = 55
```

The basic idea of the solution is that if we are given the $(n-1)$'th and $(n-2)$'th numbers, then the n'th number is trivial to compute. And assume that fib(1) and fib(2) are given, then it is trivial to calculate the fib(3). For the fib(4) we only need fib(3) and fib(2), hence we may disregard fib(1). Thus, we realize that we can cyclicly update the previous, current, and next values by shifting values until we have reached the desired fib(n). This is implement in Listing 8.7 as function `fib`, which takes an integer `n` as argument and returns the n'th Fibonacci number. The function does this iteratively using a `for`-loop, where `i` is the counter value, `pair` is the pair of the $i-1$'th and i'th Fibonacci numbers. In the body of the loop, the i'th and $i+1$'th numbers are assigned to `pair`. The `for`-loop automatically updates `i` for next iteration. When $n < 2$ then the body of the for-loop is not evaluated, and 1 is returned. This is of course wrong for $n < 1$, but we will ignore this for now.

The same program but using a while-loop is shown in Listing 8.8.

Listing 8.8 fibWhile.fsx:
Search for the largest Fibonacci number less than a specified number.

```
1  let fib (n : int) : int =
2    let mutable pair = (0, 1)
3    let mutable i = 1
4    while i < n do
5      pair <- (snd pair, fst pair + snd pair)
6      i <- i + 1
7    snd pair
8  
9  printfn "fib(1) = %d" (fib 1)
10 printfn "fib(2) = %d" (fib 2)
11 printfn "fib(3) = %d" (fib 3)
12 printfn "fib(10) = %d" (fib 10)
```

```
$ fsharpc --nologo fibWhile.fsx && mono fibWhile.exe
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(10) = 55
```

As can be seen, the program is almost identical. In this case, the for-loop is to be preferred, since more lines of code typically mean more chances of making a mistake. However, while-loops are still simpler and possibly easier to argue correctness about. To understand what is being calculated in code such as the while-loop in Listing 8.8, we can describe the loop in terms of its *loop invariant*. An *invariant* is a statement that is always true at a particular point in a program, and a loop invariant is a statement which is true at the beginning and end of a loop. In line 4 in Listing 8.8, we may state the invariant: The variable `pair` is the pair of the $i-1$'th and i'th Fibonacci numbers. This is provable by induction:

· loop invariant
· invariant

Base case: Before entering the while loop, `i` is 1, `pair` is (0, 1). Thus, the invariant is true.

Induction step: Assuming that `pair` is the $i-1$'th and i'th Fibonacci numbers, then the body first assigns a new value to `pair` as the i'th and $i+1$'th Fibonacci numbers, then increases i with one such that at the end of the loop the `pair` again contains the the $i-1$'th and i'th Fibonacci numbers. Thus, the invariant is true.

Thus when know that the second value in `pair` holds the value of the i'th Fibonacci number, and since we further may prove that when line 7 is only reached, then $i = n$, and thus that `fib` returns the n'th Fibonacci number.

While-loops also allows for other logical structures than for-loops, such as the case when the number of iteration cannot easily be decided when entering the loop. As an example, consider a slight variation of the above problem, where we wish to find the largest Fibonacci number less than some number. A solution to this problem is shown in Listing 8.9.

Listing 8.9 fibWhileLargest.fsx:
Search for the largest Fibonacci number less than a specified number.

```
let largestFibLeq n =
  let mutable pair = (0, 1)
  while snd pair <= n do
    pair <- (snd pair, fst pair + snd pair)
  snd pair

for i = 1 to 10 do
  printfn "largestFibLeq(%d) = %d" i (largestFibLeq i)
```

```
$ fsharpc --nologo fibWhileLargest.fsx && mono fibWhileLargest.exe
largestFibLeq(1) = 2
largestFibLeq(2) = 3
largestFibLeq(3) = 5
largestFibLeq(4) = 5
largestFibLeq(5) = 8
largestFibLeq(6) = 8
largestFibLeq(7) = 8
largestFibLeq(8) = 13
largestFibLeq(9) = 13
largestFibLeq(10) = 13
```

The strategy here is to iteratively calculate numbers in Fibonacci's sequence until we've found one larger than the argument `n`, and then return the previous. This could not be calculated with a for-loop.

8.2 Conditional expressions

Programs often contains code, which only should be executed under certain conditions. This can be expressed as `if`-expressions, whos syntax is as follows.

· `if`
· `then`
· `elif`
· `else`

Listing 8.10 Conditional expressions.

```
if <cond> then <expr> {elif <cond> then <expr>} [else <expr>]
```

The condition `<con>` is an expression resulting in a Boolean value, and there can be zero or more `elif` conditions as indicated by `{}`. Each expression `<expr>` is called a *branch*, and all branches must have the identical type, such that regardless which branch is chosen, then the type of the result of the conditional expression is the same. The result of the conditional expression is the first branch, for which its condition was true and if all conditions are false then the `else`-branch is evaluated. If no `else` expression is present, then "`()`" will be returned. See Listing 8.11 for a simple example.

· branch

Listing 8.11 condition.fsx:
Conditions evaluates their branches depending on the value of the condition.

```
if true then printfn "hi" else printfn "bye"
if false then printfn "hi" else printfn "bye"
```

```
$ fsharpc --nologo condition.fsx && mono condition.exe
hi
bye
```

The lightweight syntax allows for newlines entered everywhere, but indentation must be used to express scope.

To demonstrate conditional expressions, let us write a program, which writes the sentence, “I have n apple(s)”, where the plural 's' is added appropriately for various ns. This is done in Listing 8.12 using the lightweight syntax.

Listing 8.12 conditionalLightweight.fsx:
Using conditional expression to generate different strings.

```
let applesIHave n =
  if n < -1 then
    "I owe " + (string -n) + " apples"
  elif n < 0 then
    "I owe " + (string -n) + " apple"
  elif n < 1 then
    "I have no apples"
  elif n < 2 then
    "I have 1 apple"
  else
    "I have " + (string n) + " apples"

printfn "%A" (applesIHave -3)
printfn "%A" (applesIHave -1)
printfn "%A" (applesIHave 0)
printfn "%A" (applesIHave 1)
printfn "%A" (applesIHave 2)
printfn "%A" (applesIHave 10)
```

```
$ fsharpc --nologo conditionalLightWeight.fsx
$ mono conditionalLightWeight.exe
"I owe 3 apples"
"I owe 1 apple"
"I have no apples"
"I have 1 apple"
"I have 2 apples"
"I have 10 apples"
```

The sentence structure and its variants give rise to a more compact solution since the language to be returned to the user is a variant of "I have/or no/number apple(s)", i.e., under certain conditions should the sentence use “have” and “owe” etc. So, we could instead make decisions on each of these sentence parts and then built the final sentence from its parts. This is accomplished in the following example:

Listing 8.13 conditionalLightweightAlt.fsx:
Using sentence parts to construct the final sentence.

```
let applesIHave n =
  let haveOrOwe = if n < 0 then "owe" else "have"
  let pluralS = if (n = 0) || (abs n) > 1 then "s" else ""
  let number = if n = 0 then "no" else (string (abs n))

  "I " + haveOrOwe + " " + number + " apple" + pluralS

printfn "%A" (applesIHave -3)
printfn "%A" (applesIHave -1)
printfn "%A" (applesIHave 0)
printfn "%A" (applesIHave 1)
printfn "%A" (applesIHave 2)
printfn "%A" (applesIHave 10)
```

```
$ fsharpc --nologo conditionalLightWeightAlt.fsx
$ mono conditionalLightWeightAlt.exe
"I owe 3 apples"
"I owe 1 apple"
"I have no apples"
"I have 1 apple"
"I have 2 apples"
"I have 10 apples"
```

While arguably shorter, this solution is also denser, and for a small problem like this, it is most likely more difficult to debug and maintain.

Note that both `elif` and `else` branches are optional, which may cause problems. For example, both `let a = if true then 3` and `let a = if true then 3 elif false then 4` will be invalid, since F# is not smart enough to realize that the type of the expression is uniquely determined. Instead, F# looks for the `else` to ensure all cases have been covered, and that `a` always will be given a unique value of the same type regardless of the branch taken in the conditional statement, hence, `let a = if true then 3 else 4` is the only valid expression of the 3. In practice, F# assumes that the omitted branch returns "()", and thus it is fine to say `let a = if true then ()` and `if true then printfn "hej"`. Nevertheless, it is good practice in F# always to include an `else` branch.

8.3 Programming intermezzo: Automatic conversion of decimal to binary numbers

Using loops and conditional expressions, we are now able to solve the following problem:

Problem 8.2

Given an integer on decimal form, write its equivalent value on the binary form.

To solve this problem, consider odd numbers: They all have the property, that the least significant bit is 1, e.g., $1_2 = 1, 101_2 = 5$ in contrast to even numbers such as $110_2 = 6$. Division by 2 is equal to right-shifting by 1, e.g., $1_2/2 = 0.1_2 = 0.5, 101_2/2 = 10.1_2 = 2.5, 110_2/2 = 11_2 = 3$. Thus, by integer division by 2 and checking the remainder, we may sequentially read off the least significant bit. This leads to the algorithm shown in Listing 8.14.

Listing 8.14 dec2bin.fsx:
Using integer division and remainder to write any positive integer on binary form.

```
 1 let dec2bin n =
 2   if n < 0 then
 3     "Illegal value"
 4   elif n = 0 then
 5     "0b0"
 6   else
 7     let mutable v = n
 8     let mutable str = ""
 9     while v > 0 do
10       str <- (string (v % 2)) + str
11       v <- v / 2
12     "0b" + str
13
14
15 printfn "%4d -> %s" -1 (dec2bin -1)
16 printfn "%4d -> %s" 0 (dec2bin 0)
17 for i = 0 to 3 do
18   printfn "%4d -> %s" (pown 10 i) (dec2bin (pown 10 i))
```

```
1 $ fsharpc --nologo dec2bin.fsx && mono dec2bin.exe
2   -1 -> Illegal value
3    0 -> 0b0
4    1 -> 0b1
5   10 -> 0b1010
6  100 -> 0b1100100
7 1000 -> 0b1111101000
```

In the code, the states `v` and `str` are iteratively updated until `str` finally contains the desired solution.

To prove that Listing 8.14 calculates the correct sequence we use induction. First we realize, that for $v < 1$ then the `while` loop is skipped, and the result is trivially true. We will concentrate on line 9 in Listing 8.14, and we will prove the following loop invariant: The string `str` contains all the bits of `n` to the right of the bit pattern remaining in variable `v`.

Base case $n = 000\ldots000x$: If n only uses the lowest bit then $n = 0$ or $n = 1$. If $n = 0$ then it is trivially correct. Considering the case $n = 1$: Before entering into the loop, `v` is 1, `str` is the empty string, so the invariant is true. The condition of the while-loop is $1 > 0$ so execution enters the loop. Since integer division of 1 by 2 gives 0 with remainder 1, then `str` is set to `"1"` and `v` to 0. Now we reexamine the while-loop's condition, $0 > 0$, which is false, so we exit the loop. At this point, `v` is 0 and `str` is `"1"`, so all bits have been shifted from `n` to `str` and none are left in `v`. Thus the invariant is true. Finally, the program returns `"0b1"`.

Induction step: Consider the case of $n > 1$, and assume that the invariant is true when entering the loop, i.e., that m bits already have been shifted to `str` and that $n > 2^m$. In this case `v` contains the remaining bits of `n`, which is the integer division `v = n / 2**m`. Since $n > 2^m$ then `v` is non-zero and the loop conditions is true, so we enter the loop body. In the loop body we concatenate the rightmost bit of `v` to the left of `str` using `v % 2`, and right-shift `v` one bit to the right with `v <- v / 2`. Thus, when returning to the condition, the invariant is true, since the right-most bit in `v` has been shifted to `str`. This continues until all bits have been shifted to `str` and `v = 0`, in which case the loop terminates, and `"0b"+str` is returned.

Thus we have proven, that `dec2bin` correctly converts integers to strings representing binary numbers.