**Listing 8.13, dec2bin.fsx:**
**Using integer division and remainder to write any positive integer on binary form.**

```
let dec2bin n =
  let rec dec2binHelper n =
    let mutable v = n
    let mutable str = ""
    while v > 0 do
      str <- (string (v % 2)) + str
      v <- v / 2
    str

  if n < 0 then
    "Illegal value"
  elif n = 0 then
    "0b0"
  else
    "0b" + (dec2binHelper n)

printfn "%4d -> %s" -1 (dec2bin -1)
printfn "%4d -> %s" 0 (dec2bin 0)
for i = 0 to 3 do
  printfn "%4d -> %s" (pown 10 i) (dec2bin (pown 10 i))
```

```
  -1 -> Illegal value
   0 -> 0b0
   1 -> 0b1
  10 -> 0b1010
 100 -> 0b1100100
1000 -> 0b1111101000
```

Another solution is to use recursion instead of the "`while`" loop:

**Listing 8.14, dec2binRec.fsx:**
**Using recursion to write any positive integer on binary form, see also Listing 8.13.**

```
let dec2bin n =
  let rec dec2binHelper n =
    if n = 0 then ""
    else (dec2binHelper (n / 2)) + string (n % 2)

  if n < 0 then
    "Illegal value"
  else
    "0b" +
    if n = 0 then
      "0"
    else
      dec2binHelper n

printfn "%4d -> %s" -1 (dec2bin -1)
printfn "%4d -> %s" 0 (dec2bin 0)
for i = 0 to 3 do
  printfn "%4d -> %s" (pown 10 i) (dec2bin (pown 10 i))
```

```
  -1 -> Illegal value
   0 -> 0b0
   1 -> 0b1
  10 -> 0b1010
 100 -> 0b1100100
1000 -> 0b1111101000
```

Listing 8.13 is a typical imperative solution, where the states `v` and `str` are iteratively updated until `str` finally contains the desired solution. Listing 8.14 is a typical functional programming solution, to be discussed in ??, where the states are handled implicitly as new scopes created by recursively calling the helper function. Both solutions have been created using a local helper function, since both solutions require special treatment of the cases $n < 0$ and $n = 0$.

Let us compare the two solutions more closely: The computation performed is the same in both solutions, i.e., integer division and remainder is used repeatedly, but since the recursive solution is slightly shorter, then one could argue that it is better, since shorter programs typically have fewer errors. However, shorter program also typically means that understanding them is more complicated, since shorter programs often rely on realisations that the author had while programming, which may not be properly communicated by the code nor comments. Speedwise, there is little difference between the two methods: 10,000 conversions of `System.Int32.MaxValue`, i.e., the number 2,147,483,647, takes about 1.1 sec for both methods on an 2,9 GHz Intel Core i5 machine.

Notice also, that in Listing 8.14, the prefix `"0b"` is only written once. This is advantageous for later debugging and updating, e.g., if we later decide to alter the program to return a string without the prefix or with a different prefix, then we would only need to change one line instead of two. However, the program has gotten slightly more difficult to read, since the string concatenation operator and the `if` statement are now intertwined. There is thus no clear argument for preferring one over the other by this argument.

Proving that Listing 8.14 computes the correct sequence is easily done using the induction proof technique: The result of `dec2binHelper 0` is clearly an empty string. For calls to `dec2binHelper n` with $n > 0$, we check that the right-most bit is correctly converted by the remainder function, and that this string is correctly concatenated with `dec2binHelper` applied to the remaining bits. A simpler way to state this is to assume that `dec2binHelper` has correctly programed, so that in the body of `dec2binHelper`, then recursive calls to `dec2binHelper` returns the correct value. Then we only need to check that the remaining computations are correct. Proving that Listing 8.13 calculates the correct sequence essentially involves the same steps: If $v = 0$ then the "`while`" loop is skipped, and the result is the initial value of `str`. For each iteration of the "`while`" loop, assuming that `str` contains the correct conversions of the bits up till now, we check that the remainder operator correctly concatenates the next bit, and that `v` is correctly updated with the remaining bits. We finally check that the loop terminates, when no more 1-bits are left in `v`. Comparing the two proofs, the technique of assuming that the problem has been solved, i.e., that recursive calls will work, helps us focus on the key issues for the proof. Hence, we conclude that the recursive solution is most elegantly proved, and thus preferred.

Much better performance could be achieved if you avoid string concatenation. Both a low-level imperative version and a more high-level functional version can be written...

# Chapter 9

# Organising code in libraries and application programs

In this chapter we will focus on a number of ways to make it available as a *library* function in F#, and by library we mean a collection of types, values and functions that an application program can use. A library does not perform calculations on its own.

· library

F# includes several programming structures to organize code in libraries: Modules, namespaces and classes. In this chapter, we will describe modules and namespaces. Classes will be described in detail in Chapter 19.

Consider the following problem:

**Problem 9.1:**

Design a library of utility functions that may be reused in several programs. The library should as minimum include the function:

```
type floatFunction = float -> float -> float
let apply (f : floatFunction) (x : float) (y : float) : float = f x y
```

The function `apply` here serves as a dummy. [1]

## 9.1 Modules

An F# *module*, not to be confused with a Common Language Infrastructure module (see Appendix D), is a programming structure used to organise type declarations, values, functions, etc.

· module

Every implementation and script file in F# implicitly defines a module, and the module name is given by the filename. Thus, creating a script file `Meta.fsx` with the following content:

[1] Jon: **Should we introduce the concept of linking compiled code?**

**Listing 9.1 Meta.fsx:**
**A script file defining the apply function.**

```
type floatFunction = float -> float -> float
let apply (f : floatFunction) (x : float) (y : float) : float = f x y
```

we've implicitly defined a module of name Meta. Another script file may now use this function, and it is accessed using the "." notation, i.e., Meta.apply will refer to this function in other programs. A use could be:

**Listing 9.2 MetaApp.fsx:**
**Defining a script calling the module.**

```
let add : Meta.floatFunction = fun x y -> x + y
let result = Meta.apply add 3.0 4.0
printfn "3.0 + 4.0 = %A" result
```

In the above, we have explicitly used the module's type definition for illustration purposes. A shorter and possibly simpler program would have been to define add as `let add x y = x + y`, since F#'s typesystem will deduce the its implied type. However, **explicit definitions of types is recommended for readability.** Hence, an alternative to the above's use of lambda functions is, `let add (x: float) (y: float) : float = x + y`. To compile the module and the application program, we write:

Advice

**Listing 9.3: Compiling both the module and the application code. Note that fileorder matters, when compiling several files.**

```
$ fsharpc --nologo MetaApp.fsx && mono MetaApp.exe
./fsxeval2out: line 19: fsharpc: command not found
```

Notice, since the F# compiler reads through the files once, the order of the filenames in the compile command is very important. Hence, the script containing the module and function definitions must be to the left of the script containing their use. Notice also that if not otherwise specified, then the F# compiler produces an .exe file derived from the last filename in the list of filenames.

We may also explicitly define the module name using the "module" as illustrated here:

**Listing 9.4 MetaExplicit.fsx:**
**Explicit definition of the outermost module.**

```
module Meta
type floatFunction = float -> float -> float
let apply (f : floatFunction) (x : float) (y : float) : float = f x y
```

Since created a new file, where the module Meta is explicitly defined, we can use the same application program.

**Listing 9.5: Changing the module definition to explicit naming has no effect on the application nor the compile command.**

```
$ fsharpc --nologo MetaExplicit.fsx MetaApp.fsx && mono MetaApp.exe
./fsxvar2out: line 15: fsharpc: command not found
./fsxvar2out: line 16: mono: command not found
```

Note that, since `MetaExplicitModuleDefinition.fsx` explicitly defines the module name, `apply` is not available to an application program as `MetaExplicitModuleDefinition.apply`. **It is recommended that module names are defined explicitly, since filenames may change due to external conditions.** I.e., filenames are typically set from the perspective of the filesystem. The user may choose to change names to suit a filesystem structure, or different platforms may impose different file naming convention. Thus, direct linking of filenames with the internal workings of a program is a needless complication of structure.

Advice

The definitions inside a module may be accessed directly from an application program (omitting the "."-notation) by use of the "open" keyword. I.e., we can modify `MetaUse.fsx` to as follows:

**Listing 9.6 MetaAppWOpen.fsx: Avoiding the "."-notation by the "open" keyword. Beware of namespace pollution.**

```
open Meta
let add : floatFunction = fun x y -> x + y
let result = apply add 3.0 4.0
printfn "3.0 + 4.0 = %A" result
```

In this case, the namespace of our previously defined module is included into the scope of the application functions, and its types, values, functions, etc. can be used directly. ~~Thus~~ Compilation works as before:

**Listing 9.7: How the application program opens the module has no effect on the module code nor compile command.**

```
$ fsharpc --nologo MetaAppWOpen.fsx && mono MetaAppWOpen.exe
./fsxeval2out: line 19: fsharpc: command not found
```

The "open"-keyword should be used sparingly, since including a library's definitions into the application scope can cause surprising naming conflicts, since the user of a library typically has no knowledge of the inner workings of the library. E.g., the user may accidentally use code defined in the library, but with different type and functionality than intended, which the typesystem will use to deduce types in the application program, and therefore will either give syntax or runtime errors that are difficult to understand. This is known as *namespace pollution*, and for clarity **it is recommended to use the "open"-keyword sparingly**. Note that, for historical reasons, the work namespace pollution is used to cover both pollution due to modules and namespaces.

· namespace pollution
Advice

Modules may also be nested, in which case the nested definitions must use the "="-sign and must be appropriately indented. Here is an example of a nested module containing two submodules:

**Listing 9.8 nestedModules.fsx:**
**Modules may be nested.**

```
module Utilities
  let PI = 3.1415
  module Meta =
    type floatFunction = float -> float -> float
    let apply (f : floatFunction) (x : float) (y : float) : float = f x y
  module MathFcts =
    let add : Meta.floatFunction = fun x y -> x + y
```

In this case, `Meta` and `MathFcts` are defined ~~on~~ at the same level and said to be siblings, while `Utilities` is defined ~~at~~ a higher level. In this relation, the former two are said to be the children of the latter. Note that the nesting respects the lexical scope rules, such that the constant `PI` is directly accessible in both modules `Meta` and `MathFcts`, as is the module `Meta` in `MathFcts` but not `MathFcts` in `Meta`. The "."-notation is reused to index deeper into the module hierarchy as the following example shows.

**Listing 9.9 nestedModulesApp.fsx:**
**Applications using nested modules require additional usage of the "." notation to navigate the nesting tree.**

```

let add : Utilities.Meta.floatFunction = fun x y -> x + y
let result = Utilities.Meta.apply Utilities.MathFcts.add 3.0 4.0
printfn "3.0 + 4.0 = %A" result
```

Modules can be recursive using the "`rec`"-keyword, meaning that in our example we can make the outer module recursive as follows.[2]

**Listing 9.10 nestedRecModules.fsx:**
**Mutual dependence on nested modules requires the "`rec`" keyword in the module definition.**

```
module rec Utilities
  module Meta =
    type floatFunction = float -> float -> float
    let apply (f : floatFunction) (x : float) (y : float) : float = f x y
  module MathFcts =
    let add : Meta.floatFunction = fun x y -> x + y
```

The consequence is that the modules `Meta` and `MathFcts` are accessible in both modules, but compilation will now give a warning, since soundness of the code will first be checked at runtime. In general it is advised to **avoid programming constructions, whose validity cannot be checked at compile-time.** Advice

## 9.2 Namespaces

An alternative structure ~~to~~ modules is a *namespace*, which only can hold modules and type declarations and only works in compiled mode. Namespaces are defined as explicitly defined outer modules, ~~e.g.,~~ · namespace

[2] Jon: **Dependence on version 4.1 and higher.**

**Listing 9.11 namespace.fsx:**
**Defining a namespace is similar to explicitly named modules.**

```
1 namespace Utilities
2 type floatFunction = float -> float -> float
3 module Meta =
4   let apply (f : floatFunction) (x : float) (y : float) : float = f x y
```

Note that when putting code in a namespace, then the first line of the file (other than comments and compiler directives) must be the one starting with `namespace`.

As for modules, the content of a namespace is accessed using the "." notation.

**Listing 9.12 namespaceApp.fsx:**
**The "."-notation lets the application program accessing functions and types in a namespace.**

```
1 let add : Utilities.floatFunction = fun x y -> x + y
2 let result = Utilities.Meta.apply add 3.0 4.0
3 printfn "3.0 + 4.0 = %A" result
```

Likewise, compilation is performed identically.

**Listing 9.13: Compilation of files including namespace definitions uses the same procedure as modules.**

```
1 $ fsharpc --nologo namespaceApp.fsx && mono namespaceApp.exe
2 ./fsxeval2out: line 19: fsharpc: command not found
```

Hence, from an application point of view, it is not immediately possible to see, that `Utilities` is defined as a namespace and not a module. However, in contrast to modules, namespaces may span several files. E.g., we may add a third file containing extending the `Utilities` namespace with the `MathFcts` module as demonstrated below.

**Listing 9.14 namespaceExtension.fsx:**
**Namespaces may span several files. Here is shown an extra file, which extends the `Utilities` namespace.**

```
1 namespace Utilities
2 module MathFcts =
3   let add : floatFunction = fun x y -> x + y
```

To compile we now need to include all three files in the right order. Likewise, compilation is performed identically.

**Listing 9.15: Compilation of namespaces defined in several files requires careful consideration of order, since the compiler reads once and only once through the files in the order they are given.**

```
$ fsharpc --nologo namespace.fsx namespaceExtension.fsx namespaceApp.fsx
   && mono namespaceApp.exe
./fsxvar2out: line 15: fsharpc: command not found
./fsxvar2out: line 16: mono: command not found
```

The order matters since `namespaceExtension.fsx` relies on the definition of `floatFunction` in `namespace.fsx`. You can use extensions to extend existing namespaces included with the F# compiler.[3][4]

Namespaces may also be nested. In contrast to modules, nesting defined using the "." notation, i.e., to create a child namespace `more` of `Utilities` we must use initially write `namespace Utilities.more`. Indentation is ignored in the `namespace` line, thus left-most indentation is almost always used. Namespaces observed lexical scope, and identically to modules, namespaces containing mutually dependent children can be declared using the "`rec`" keyword, e.g., `namespace rec Utilities`.

Libraries may be distributed in compile form as `.dll` files. This saves the use for having recompile a possibly large library every time an application program needs it. In order to produce a library file from `MetaExplicitModuleDefinition.fsx` and then compile an application program, we first use the compiler's `-a` option to produce the `.dll`.

**Listing 9.16: A stand-alone `.dll` file is created and used with special compile commands.**

```
$ fsharpc --nologo -a MetaExplicit.fsx
./fsxeval2out: line 19: fsharpc: command not found
```

This produces the file `MetaExplicit.dll`, which may be linked to an application using the `-r` option during compilation.[5]

**Listing 9.17: The library is linked to an application during compiliation to produce runnable code.**

```
$ fsharpc --nologo -r MetaExplicit.dll MetaApp.fsx && mono MetaApp.exe

error FS0078: Unable to find the file 'MetaExplicit.dll' in any of
 /usr/local/Cellar/mono/5.0.1.1/lib/mono/4.5
 /usr/local/Cellar/mono/5.0.1.1/lib/mono/4.5/Facades
 /usr/local/Cellar/mono/5.0.1.1/lib/mono/4.5-api
 /usr/local/Cellar/mono/5.0.1.1/lib/mono/4.5-api/Facades
 /Users/mael/svn/fsharpNotes/src
 /usr/local/Cellar/mono/5.0.1.1/lib/mono/fsharp
```

Libraries can of course be a compilation of any number of files into a single `.dll` file. `.dll`-files may be loaded dynamically in script files (`.fsx`-files) using the *`#r` directive* as illustrated below.

· `#r` directive

[3] Jon: **Something about intrinsic and optional extension** `https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/type-extensions`.

[4] Jon: **Perhaps something about the global namespace** `global`.

[5] Jon: **This is the MacOS option standard, Windows is slightly different.**

**Listing 9.18 MetaHashApp.fsx:**
**The `.dll` file may be loaded dynamically in `.fsx` script files and in interactive mode. Nevertheless, this usage is not recommended.**

```
#r "MetaExplicit.dll"
let add : Meta.floatFunction = fun x y -> x + y
let result = Meta.apply add 3.0 4.0
printfn "3.0 + 4.0 = %A" result
```

We may now omit the explicit mentioning of the library when compiling.

**Listing 9.19: When using the `#r` directive, then the `.dll` file need not be explicitly included in the list of files to be compiled.**

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

MetaHashApp.fsx(1,1): error FS0078: Unable to find the file 'MetaExplicit
   .dll' in any of
 /usr/local/Cellar/mono/5.0.1.1/lib/mono/4.5
 /usr/local/Cellar/mono/5.0.1.1/lib/mono/4.5/Facades
 /usr/local/Cellar/mono/5.0.1.1/lib/mono/4.5-api
 /usr/local/Cellar/mono/5.0.1.1/lib/mono/4.5-api/Facades
 /Users/mael/svn/fsharpNotes/src
 /usr/local/Cellar/mono/5.0.1.1/lib/mono/fsharp
```

The `#r` directive is also used to include a library in interactive mode. However, for code to be compiled, the use of the `#r` directive requires that the filesystem path to the library is coded inside the script. As for module names, direct linking of filenames with the internal workings of a program is a needless complication of structure, and **it is recommended not to rely on the use of the `#r` directive.** Advice

In the above, we have compiled *script files* into libraries. However, F# has reserved the `.fs` filename suffix for library files, and such files are called *implementation files*. In contrast to script files, implementation files do not support the `#r` directive. When compiling a list of implementation and script files, all but the last file must explicitly define a module or a namespace.

· script files
· implementation files

Both script and implementation files may be augmented with *signature files*. A signature file contains no implementation but only type definitions. Signature files offers three distinct features:

· signature files

1. Signature files can be used as part of the documentation of code, since type information is of paramount importance for an application programmer to use a library.
2. Signature files may be written before the implementation file. This allows for a higher level programming design, one that focusses on *which* functions should be included and *how* they should be pieced together. For large programs this is a better approach to programming, than just hitting the keyboard happily typing away with only a loose plan.
3. Signature files allow for access control. Most importantly, if the type definitions not available in the signature file, it is not available to the application program. They are thus private and can only be used internally in the library code. More fine grained control is available relating to classes, and will be discussed in Chapter 19.

Signature files can be generated automatically using the `--sig:<filename>` compiler directive. To demonstrate this, consider the following implementation file:

**Listing 9.20 MetaWAdd.fs:**
**An implementation file including the add function.**

```
1 module Meta
2 type floatFunction = float -> float -> float
3 let apply (f : floatFunction) (x : float) (y : float) : float = f x y
4 let add (x : float) (y : float) : float = x + y
```

A signature file may be automatically generated as follows.

**Listing 9.21: Automatic generation of a signature file at compile time.**

```
1 $ fsharpc --nologo --sig:MetaWAdd.fsi MetaWAdd.fs
2 ./fsxeval2out: line 19: fsharpc: command not found
```

The warning can safely be ignored, since it is at this point not our intention to produce runnable code. The above has generated the following signature file.

**Listing 9.22 MetaWAdd.fsi:**
**An automatically generated signature file from MetaWAdd.fs.**

```
1 module Meta
2 type floatFunction = float -> float -> float
3 val apply : f:floatFunction -> x:float -> y:float -> float
4 val add : x:float -> y:float -> float
```

We can generate a library using the automatically generated signature file using `fsharpc -a MetaWAdd .fsi MetaWAdd.fs`, ~~however this~~ which is identical to compiling the `.dll` file without the signature file. However, if we remove, e.g., the type definition for `add` in the signature file, then this function becomes private to the module, and cannot be accessed outside. Hence, using

**Listing 9.23 MetaWAddRemoved.fsi:**
**Removing the type defintion for add from MetaWAdd.fsi.**

```
1 module Meta
2 type floatFunction = float -> float -> float
3 val apply : f:floatFunction -> x:float -> y:float -> float
```

and recompiling the `.dll`

**Listing 9.24: Automatic generation of a signature file at compile time.**

```
1 $ fsharpc --nologo -a MetaWAddRemoved.fsi MetaWAdd.fs
2 ./fsxeval2out: line 19: fsharpc: command not found
```

generates no error. But when using the newly created `MetaWAdd.dll`

rephrase to avoid code blocks to be parts of a sentence.

**Listing 9.25: Automatic generation of a signature file at compile time.**

```
$ fsharpc --nologo -r MetaWAdd.dll MetaWAddApp.fs

MetaWAddApp.fs(2,25): error FS0039: The value or constructor 'add' is not
    defined.
```

we get a syntax error, since `add` now is inaccessible to the application program. results in an error.

6

[6] Jon: **Possible highlight changes in code stumps, and possibly improve reference to code used fore each compilation example.**