Learning to program with F#

Jon Sporring & Torben Mogensen

Department of Computer Science,
University of Copenhagen

July 19, 2017

Contents

Chapter 3

Executing F# code

3.1 Source code

F# is a functional first programming language, meaning that it has strong support for functional programming, but F# also supports imperative and object oriented programming. It also has strong support for parallel programming and information rich programs. It was originally developed for Microsoft's .Net platform, but is available as open source for many operating systems through Mono. In this text we consider F# 4.0 and its Mono implementation, which is different from .Net mainly in terms of the number of libraries accessible. The complete language specification is described in `http://fsharp.org/specs/language-spec/4.0/FSharpSpec-4.0-latest.pdf`.

F# has 2 modes of execution, *interactive* and *compiled*. Interactive mode is well suited for small experiments or back-of-an-envelope calculations, but not for programming in general. In Mono, the interactive system is started by calling `fsharpi` from the *console*, while compilation is performed with `fsharpc` and execution of the compiled code is performed using the `mono` command. The various forms of fsharp programs are identified by suffixes:

· interactive
· compiled
· console

`.fs` An *implementation file*, e.g., `myModule.fs`

`.fsi` A *signature file*, e.g., `myModule.fsi`

`.fsx` A *script file*, e.g., `gettingStartedStump.fsx`

`.fsscript` Same as `.fsx`, e.g., `gettingStartedStump.fsscript`

`.exe` An *executable file*, e.g., `gettingStartedStump.exe`

· implementation file
· signature file
· script file
· executable file

The implementation, signature, and script files are all typically compiled to produce an executable file, but syntactical correct code can also be entered into the interactive system, in which case these are called *script-fragments*. The implementation and signature files are special kinds of script files used for building *modules*. Modules are collections of smaller programs used by other programs, which will be discussed in detail in Part IV.

· script-fragments
· modules

3.2 Executing programs

Programs may either be executed by the interpreter or by compiling and executing the compiled code.

In Mono the interpreter is called `fsharpi` and can be used in 2 ways: interactively, where a user enters 1 or more script-fragments separated by the "`;;`" lexeme, or to execute a script file treated as a single script-fragment. To illustrate the difference, consider the following program, which declares a value `a` to be the decimal value 3.0 and finally print it to the console:

Listing 3.1 gettingStartedStump.fsx:
A simple demonstration script.

```
let a = 3.0
printfn "%g" a
```

An interactive session is obtained by starting the console, typing the `fsharpi` command, typing the lines of the program, and ending the script-fragment with the "`;;`" lexeme. The following dialogue demonstrates the workflow, where what the user types has been highlighted by a box:

Listing 3.2: An interactive session.

```
$ fsharpi

F# Interactive for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License

For help type #help;;

> let a = 3.0
- printfn "%g" a;;
3

val a : float = 3.0
val it : unit = ()

> #quit;;
```

The interpreter is stopped by pressing `ctrl-d` or typing `"#quit;;"`. Conversely, executing the file with the interpreter as follows,

Listing 3.3: Using the interpreter to execute a script.

```
$ fsharpi gettingStartedStump.fsx
3
```

Finally, compiling and executing the code is performed as,

Listing 3.4: Compiling and executing a script.

```
$ fsharpc gettingStartedStump.fsx
F# Compiler for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
$ mono gettingStartedStump.exe
3
```

Both the interpreter and the compiler translates the source code into a format, which can be executed by the computer. While the compiler performs this translation once and stores the result in the executable file, the interpreter translates the code every time the code is executed. Thus, to run the program again with the interpreter, then it must be retranslated as "`$fsharpi gettingStartedStump.fsx`". In contrast, compiled code does not need to be recompiled to be run again, only re-executed using "`$ mono gettingStartedStump.exe`".On a Macbook Pro, with a 2.9 Ghz Intel Core i5, the time the various stages takes for this script are:

Command	Time
`fsharpi gettingStartedStump.fsx`	1.88s
`fsharpc gettingStartedStump.fsx`	1.90s
`mono gettingStartedStump.exe`	0.05s

I.e., executing the script with `fsharpi` is slightly faster than by first compiling it with `fsharpc` and then executing the result with `mono`, $1.88\text{s} < 0.05\text{s} + 1.90\text{s}$, if the script were to be executed only once, but every future execution of the script using the compiled version requires only the use of `mono`, which is much faster than `fsharpi`, $1.88\text{s} \gg 0.05\text{s}$.

The interactive session results in extra output on the *type inference* performed, which is very useful for *debugging* and development of code-fragments, but both executing programs with the interpreted directly from a file and compiling and executing the program is much preferred for programming complete programs, since the starting state is well defined, and since this better supports *unit-testing*, which is a method for debugging programs.

· type inference

· debugging

· unit-testing

Part IV

Structured programming

Chapter 19

Modules and Namespaces

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 20.

Consider the following problem:

Problem 19.1:

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.

An F# *module*, not to be confused with a Common Language Infrastructure module see Chapter C, 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:

Listing 19.1 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 access using the "`.`" notation, i.e., `Meta.apply` will refer to this function in other programs. A use could be:

Listing 19.2 MetaUse.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 19.3: Compiling both the module and the application code. Note that file-order matters, when compiling several files.

```
$ fsharpc Meta.fsx MetaUse.fsx
F# Compiler for F# 4.1
Freely distributed under the Apache 2.0 Open Source License
$ mono MetaUse.exe
3.0 + 4.0 = 7.0
```

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 19.4 MetaExplicitModuleDefinition.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 19.5: Changing the module definition to explicit naming has no effect on the application nor the compile command.

```
$ fsharpc MetaExplicitModuleDefinition.fsx MetaUse.fsx
F# Compiler for F# 4.1
Freely distributed under the Apache 2.0 Open Source License
$ mono MetaUse.exe
3.0 + 4.0 = 7.0
```

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 access directly from an application program omitting the "`.`"-notation by use of the "`open`" keyword. I.e., we can modify `MetaUse.fsx` to

Listing 19.6 MetaUseWOpen.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 define module is included into the scope of the application functions, and its types, values, functions etc. can be used directly. Thus

Listing 19.7: How the application program opens the module has no effect on the module code nor compile command.

```
$ fsharpc MetaExplicitModuleDefinition.fsx MetaUseWOpen.fsx
F# Compiler for F# 4.1
Freely distributed under the Apache 2.0 Open Source License
$ mono MetaUseWOpen.exe
3.0 + 4.0 = 7.0
```

The "`open`"-keyword should 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 run-time 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 indendet.

Listing 19.8 nestedModules.fsx:
Modules may be nested.

```fsharp
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 the same level and said to be siblings, while `Utilities` is defined on 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 19.9 nestedModulesUse.fsx:
Applications using nested modules require additional usage of the "." notation to navigate the nesting tree.

```fsharp

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.[1]

Listing 19.10 nestedRecModules.fsx:
Mutual dependence on nested modules requires the "rec" keyword in the module definition.

```fsharp
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 run-time. In general it is advised to **avoid programming constructions, whose validity cannot be checked at compile-time.** Advice

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

[1] Todo: **Dependence on version 4.1 and higher.**

Listing 19.11 namespace.fsx:
Defining a namespace is similar to explicitly named modules.

```
namespace Utilities
type floatFunction = float -> float -> float
module Meta =
  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 19.12 namespaceUse.fsx:
The "."-notation lets the application program accessing functions and types in a namespace.

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

Likewise, compilation is performed identically.

Listing 19.13: Compilation of files including namespace definitions uses the same procedure as modules.

```
$ fsharpc namespace.fsx namespaceUse.fsx
F# Compiler for F# 4.1
Freely distributed under the Apache 2.0 Open Source License
$ mono namespaceUse.exe
3.0 + 4.0 = 7.0
```

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

```
namespace Utilities
module MathFcts =
  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 19.15: 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 namespace.fsx namespaceExtension.fsx namespaceUse.fsx
F# Compiler for F# 4.1
Freely distributed under the Apache 2.0 Open Source License
$ mono namespaceUse.exe
3.0 + 4.0 = 7.0
```

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.[2]

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`, and the `-r` option to compile the application program with the newly created library.[3]

Listing 19.16: A stand-alone `.dll` file is created and used with special compile commands.

```
$ fsharpc -a MetaExplicitModuleDefinition.fsx
F# Compiler for F# 4.1
Freely distributed under the Apache 2.0 Open Source License
$ fsharpc -r MetaExplicitModuleDefinition.dll MetaUse.fsx
F# Compiler for F# 4.1
Freely distributed under the Apache 2.0 Open Source License
$ mono MetaUse.exe
3.0 + 4.0 = 7.0
```

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

Listing 19.17 MetaUseHash.fsx:
The `.dll` file may be loaded dynamically in `.fsx` script files and in interactive mode. Nevertheless, this usage is not recommended.

```
#r "MetaExplicitModuleDefinition.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
```

[2]Todo: **Perhaps something about the global namespace `global`.**
[3]Todo: **This is the MacOS option standard, Windows is slightly different.**

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

Listing 19.18: When using the `#r` directive, then the `.dll` file need not be explicitly included in the list of files to be compiled.

```
$ fsharpc MetaUseHash.fsx
F# Compiler for F# 4.1
Freely distributed under the Apache 2.0 Open Source License
$ mono MetaUseHash.exe
3.0 + 4.0 = 7.0
```

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

· signature files

Listing 19.19 MetaWAdd.fs:
An implementation file including the `add` function.

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

A signature file may be automatically generated as follows.

Listing 19.20: Automatic generation of a signature file at compile time.

```
$ fsharpc --sig:MetaWAdd.fsi MetaWAdd.fs
F# Compiler for F# 4.1
Freely distributed under the Apache 2.0 Open Source License

MetaWAdd.fs(4,48): warning FS0988: Main module of program is empty:
    nothing will happen when it is run
```

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

Listing 19.21 MetaWAdd.fsi:
An automatically generated signature file from `MetaWAdd.fs`.

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

This offers three distinct features:

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 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 20.

Chapter 8

To Dos

- Remove EBNF from main body of the text, possibly extend the appendix
- Add appendix on regular expressions
- Add Torben's notes on functional programming
- Rewrite list chapter (add sequences?)
- Add a chapter comparing the 3 paradigms
- Write structured programming part
- Write chapter on pattern matching (if not already in Torben's notes)
- Move modules and namespaces earlier
- Should we add something about assemblies (`https://msdn.microsoft.com/en-us/library/hk5f40ct(v=vs.90).aspx`, `https://msdn.microsoft.com/en-us/library/ms973231.aspx`, `https://stackoverflow.com/questions/2972732/what-are-net-assemblies`)
- Add something on piping (if not already in Torben's notes)
- Add abstraction of computer: places <-> memory/disk. Mutable objects are abstractions of places `https://www.infoq.com/presentations/Value-Values`. Facts does not rime with set and get.
- Hickey: Difference between syntax and semantics. Values or locations, add a good figure. Functional programming: All values are freely shareable.
- something about organising stuff: `https://fsharpforfunandprofit.com/posts/organizing-functions/`, `https://fsharpforfunandprofit.com/posts/recipe-part3/`

Bibliography

[1] M. Auer, J. Poelz, A. Fuernweger, L. Meyer, and T. Tschurtschenthaler. Umlet, free uml tool for fast uml diagrams. `http://www.umlet.com`, Present version 14.2, version 1.0 was released June 21, 2002.

[2] Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932.

[3] European Computer Manufacturers Association (ECMA). Standard ecma-335, common language infrastructure (cli). `http://www.ecma-international.org/publications/standards/Ecma-335.htm`.

[4] International Organization for Standardization. Iso/iec 23271:2012, common language infrastructure (cli). `https://www.iso.org/standard/58046.html`.

[5] Object Management Group. Uml version 2.0. `http://www.omg.org/spec/UML/2.0/`.

[6] Programming Research Group. Specifications for the ibm mathematical formula translating system, fortran. Technical report, Applied Science Division, International Business Machines Corporation, 1954.

[7] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960.

[8] X3: ASA Sectional Committee on Computers and Information Processing. American standard code for information interchange. Technical Report ASA X3.4-1963, American Standards Association (ASA), 1963. `http://worldpowersystems.com/projects/codes/X3.4-1963/`.

[9] George Pólya. *How to solve it*. Princeton University Press, 1945.

Index