1 Executing F# Code

1.1 Source Code

F# is a functional first programming language, meaning that it has strong support for functional programming, but also supports imperative and object-oriented programming.

F# has two modes of execution, *interactive* and *compiled*. Interactive mode allows the user to interact with F# as a dialogue: The user writes statements, and F# responds immediately. Interactive mode is well suited for small experiments or back-of-an-envelope calculations, but not for programming in general. In compile mode, the user writes a complete program, which is translated or compiled using the F# compiler into a compiled file. The compiled file can be run or executed as a stand-alone program using a virtual machine called `mono`. In both the interactive and compile mode, F# statements are translated into something that can be executed on the computer. A major difference is that in interactive mode, the translation is performed everytime the program is executed, while in compiled mode the translation is performed only once.

Both interactive and compile modes can be accessed via the *console*, see **??** for more information on the console. The interactive system is started by calling `fsharpi` at the command prompt in the console, while compilation is performed with `fsharpc`, and execution of the compiled code is performed using the `mono` command.

F# programs come in many forms, which are identified by suffixes. The *source code* is an F# program written in human-readable form using an editor. F# recognizes the following types of source code files:

`.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`

Compiled code is source code translated into a machine-readable language, which can be executed by a machine. Compiled F# code is either:

`.dll` A *library file*, e.g., `myModule.dll`

`.exe` A stand-alone *executable file*, e.g., `gettingStartedStump.exe`

The implementation, signature, and script files are all typically compiled to produce an executable file, in which case they are called *scripts*, but 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 libraries. Libraries in F# are called modules, and they are collections of smaller programs used by other programs, which will be discussed in detail in **??**.

1.2 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 two ways: interactively, where a user enters one or more script-fragments separated by the "`;;`" characters, or to execute a script file treated as a single script-fragment.

To illustrate the difference between interactive and compile mode, consider the program in Listing 1.1. The code declares a value `a` to be the decimal value 3.0 and

Listing 1.1 gettingStartedStump.fsx: A simple demonstration script.

```
1 let a = 3.0
2 do printfn "%g" a
```

finally prints it to the console. The `do printfn` is a statement for displaying the content of a value to the screen, and `"%g"` is a special notation to control how the value is printed. In this case, it is printed as a decimal number. This and more will be discussed at length in the following chapters. For now, we will concentrate on how to interact with the F# interpreter and compiler.

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 dialogue in Listing 1.2 demonstrates the workflow. What the user types has been highlighted by a box.

Listing 1.2: An interactive session.

```
$ fsharpi

F# Interactive for F# 4.1 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License

For help type #help;;

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

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

> #quit;;
```

We see that after typing `fsharpi`, the program starts by stating details about itself. Then F# writes `>` indicating that it is ready to receive commands. The user types `let a = 3.0` and presses `enter`, to which the interpreter responds with `-`. This indicates that the line has been received, that the script-fragment is not yet completed, and that it is ready to receive more input. When the user types `do printfn "%g" a;;` followed by `enter`, then by "`;;`" the interpreter knows that the script-fragment is completed, it interprets the script-fragment, responds with `3` and some extra information about the entered code, and with `>` to indicate that it is ready for more script-fragments. The interpreter is stopped when the user types `#quit;;`. It is also possible to stop the interpreter by typing `ctrl-d`.

The interactive session results in extra output on the *type inference* performed. In Listing 1.2 F# states that the name `a` has *type* `float` and the value `3.0`. Likewise, the `do` statement F# refers to by the name `it`, and it has the type `unit` and value `()`. Types are very important to F# since they define how different program pieces fit together like lego bricks. They are a key ingrediens for finding errors in programs, also known as *debugging*, and much of the rest of this book is concerned with types.

Instead of running `fsharpi` interactively, we can write the script-fragment from Listing 1.1 into a file, here called `gettingStartedStump.fsx`. This file can be interpreted directly by `fsharpi` as shown in Listing 1.3.

Listing 1.3: Using the interpreter to execute a script.

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

Notice that in the file, "`;;`" is optional. We see that the interpreter executes the code and prints the result on screen without the extra type information as compared to Listing 1.2.

Finally, the file containing Listing 1.1 may be compiled into an executable file with the program `fsharpc`, and run using the program `mono` from the console. This is demonstrated in Listing 1.4.

Listing 1.4: Compiling and executing a script.

```
$ fsharpc gettingStartedStump.fsx
F# Compiler for F# 4.1 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License
$ mono gettingStartedStump.exe
3
```

The compiler reads `gettingStartedStump.fsx` and makes `gettingStartedStump.exe`, which can be run using `mono`.

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, 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 take 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`

($1.88\text{s} \gg 0.05\text{s}$).

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. Thus, **prefer compiling over interpretation.** ⋆