Learning to program with F#

Jon Sporring

August 13, 2016

Contents

Chapter 12

Input and Output

An important part of programming is handling data. A typical source of data are hard-coded bindings and expressions from libraries or the program itself, and the result is often shown on a screen either as text output on the console. This is a good starting point, when learning to program, and one which we have relied heavily upon in this book until now. However, many programs require more: We often need to ask a user to input data via, e.g., typing text on a keyboard, clicking with a mouse, striking a pose in front of a camera. We also often need to load and save data to files, retrieve and deposit information from the internet, and visualize data as graphically, as sounds, or by controlling electrical appliances. Graphical user interfaces will be discussed in Chapter 13, and here we will concentrate on working with the console, with files, and with the general concept of streams.

File and stream input and output are supported via libraries built-in classes. The `printf` family of functions is defined in the `.Printf` module of the `Fsharp.Core'` namespace, and it was discussed in Chapter **??**, and will not be discussed here. What we will concentrate on is interaction with the console through the `System.Console` class and the `System.IO` namespace.

A *file* on a computer is a resource used to store data in and retrieve data from. Files are often associated with a physical device, such as a harddisk, but can also be a virtual representation in memory. Files are durable, such that other programs can access them independently, given certain rules for access. A file has a name, a size, and a type, where the type is related to the basic unit of storage such as characters, bytes, and words, (`char`, `byte`, and `int32`). Often data requires a conversion from the internal format to and from the format stored in the file. E.g., floating point numbers are sometimes converted to ASCII using `fprintf` in order to store them to file in a human readable form, and interpreted from ASCII when retrieving them at a later point from file. Files have a low-level structure and representation, which varies from device to device, and the low-level details are less relevant for the use of the file, and most often hidden for the user. Basic operations on files are creation, opening, reading from, writing to, closing, and deleting files.

· file

A *stream* is similar to files in that they are used to store data in and retrieve data from, but streams only allow for handling of data one element at a time like the readout of a thermometer: we can make temperature readings as often as we like, producing a history of temperatures, but we cannot access the future. Hence, streams are in principle without an end, and thus have infinite size, and data from streams are programmed locally by considering the present and previous elements, while data from files may be considered a stream but also allow for global operations on all the file's data.

· stream

12.1 Interacting with the console

[1] From a programming perspective, then the console is a stream: The program may send new data to the console, but cannot return to previously sent data and make changes. Likewise, the program may retrieve input from the user, but cannot go back and ask the user to have inputted something else. The console uses 3 built-in streams in `System.Console`,,,

[1] Todo: **Spec-4.0 Section 18.2.9**

Stream	Description
`stdout`	Standard output stream used by `printf` and `printfn`.
`stderr`	Standard error stream used to display warnings and errors by Mono.
`stdin`	Standard input stream used to read keyboard input.

On the console, the standard output and error streams are displayed as text, and it is typically not possible to see a distinction between them. However, command-line interpreters such as Bash can, and it is possible from the command-line to filter output from programs according to these streams. However, a further discussion on this is outside the scope of this text. In `System.Console` there are many functions supporting interaction with the console, and the most important ones are,

Function	Description
`Write`	Write to the console. E.g., `System.Console.Write "Hello world.''`.
`WriteLine`	As `Write` but followed by newline, e.g., `System.Console.WriteLine "Hello world."`.
`Read`	Read the next key from the keyboard blocking execution as long, e.g., `System.Console.Read ()`.
`ReadKey`	As `Read` but writing the key to the console as well, e.g. , `System.Console.ReadKey ()`.
`ReadLine`	Read the next sequence of characters until newline from the keyboard, e.g. , `System.Console.ReadLine ()`.

Notice that you must supply the empty argument `()`, in order to run most of the functions instead of referring to them as values. Note also, that

```
System.Console.WriteLine "To perform the multiplication of a and b"
System.Console.Write "Enter a: "
let a = float (System.Console.ReadLine ())
System.Console.Write "Enter b: "
let b = float (System.Console.ReadLine ())
System.Console.WriteLine ("a * b = " + string (a * b))
```

Listing 12.1: Interacting with a user with `ReadLine` and `WriteLine`.

An example dialogue is,

```
To perform the multiplication of a and b
Enter a: 2.3
Enter b: 4.5
a * b = 10.35
```

The `Write` functions has less functionality than the `printf` family, and *for writing to the console,* Advice *`printf` is to be preferred.*

12.2 Storing and retriving data from a file

A file stored on the filesystem has a name, and it must be opened before it can be accessed. However, since data may have been stored in files in various ways, as part of the opening process, we must specify low-level information about how the data is to be interpreted, when F# will read it. Hence, there is a family of open functions, all residing in the `System.IO.File` class,

Function	Description
`Open`	Write to the console. E.g., `System.Console.Write "Hello world.''`.
`OpenRead`	As `Write` but followed by newline, e.g., `System.Console.WriteLine "Hello world."`.
`OpenText`	Read the next key from the keyboard blocking execution as long, e.g., `System.Console.Read ()`.
`OpenWrite`	As `Read` but writing the key to the console as well, e.g. , `System.Console.ReadKey ()`.
`ReadLine`	Read the next sequence of characters until newline from the keyboard, e.g. , `System.Console.ReadLine ()`.

[2]

family of functions.

Function	Description
File	
Directory	
Path	
System.Console.OpenStandardOutput	
System.Console.OpenStandardError	
System.Console.OpenStandardInput	
StreamReader	
StreamWriter	
MemoryStream	

```
let getAFileName () =
  let mutable filename = Unchecked.defaultof<string>
  let mutable fileExists = false
  while not(fileExists) do
    System.Console.Write("Enter Filename: ")
    filename <- System.Console.ReadLine()
    fileExists <- System.IO.File.Exists filename
  filename

let listOfFiles = System.IO.Directory.GetFiles(".")
printfn "Directory contains: %A" listOfFiles
let filename = getAFileName ()
printfn "You typed: %s" filename
```

```
let rec printFile (reader : System.IO.StreamReader) =
  if not(reader.EndOfStream) then
    let line = reader.ReadLine ()
    printfn "%s" line
    printFile reader

let filename = "readFile.fsx"
let reader = System.IO.File.OpenText filename
printFile reader
```

```
let rec printFile (reader : System.IO.StreamReader) =
  if not(reader.EndOfStream) then
    let line = reader.ReadLine ()
    printfn "%s" line
    printFile reader

let filename = "readFile.fsx"
let reader = System.IO.File.OpenText filename
printFile reader
```

Listing 12.2: readFile.fsx -

```
let rec readFile (stream : System.IO.StreamReader) =
  if not(stream.EndOfStream) then
    (stream.ReadLine ()) :: (readFile stream)
  else
    []
```

[2]Todo: **See** https://msdn.microsoft.com/en-us/library/ms404278(v=vs.110).aspx

```
let rec writeFile (stream : System.IO.StreamWriter) text =
  match text with
  | (l : string) :: ls ->
    stream.WriteLine l
    writeFile stream ls
  | _ -> ()

let reverseString (s : string) =
  System.String(Array.rev (s.ToCharArray()))

let inputStream = System.IO.File.OpenText "reverseFile.fsx"
let text = readFile inputStream
let reverseText = List.map reverseString (List.rev text)
let outputStream = System.IO.File.CreateText "xsf.eliFesrever"
writeFile outputStream reverseText
outputStream.Close ()
printfn "%A" reverseText
```

```
["txeTesrever "A%" nftnirp"; ")( esolC.maertStuptuo";
 "txeTesrever maertStuptuo eliFetirw";
 ""reverseFile.fsx" txeTetaerC.eliF.OI.metsyS = maertStuptuo tel";
 ")txet ver.tsiL( gnirtSesrever pam.tsiL = txeTesrever tel";
 "maertStupni eliFdaer = txet tel";
 ""xsf.eliFesrever" txeTnepO.eliF.OI.metsyS = maertStupni tel"; "";
 ")))(yarrArahCoT.s( ver.yarrA(gnirtS.metsyS  ";
 "= )gnirts : s( gnirtSesrever tel"; ""; ")( >- _ |  ";
 "sl maerts eliFetirw    "; "l eniLetirW.maerts    ";
 ">- sl :: )gnirts : l( |  "; "htiw txet hctam  ";
 "= txet )retirWmaertS.OI.metsyS : maerts( eliFetirw cer tel"; ""; "][    ";
 "esle  "; ")maerts eliFdaer( :: ))( eniLdaeR.maerts(    ";
 "neht )maertSfOdnE.maerts(ton fi  ";
 "= )redaeRmaertS.OI.metsyS : maerts( eliFdaer cer tel"]
```

Listing 12.3: reverseFile.fsx -

Bibliography

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

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

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

[4] 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/`.

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

Index