Learning to program with F#

Jon Sporring

October 31, 2016

Contents

Chapter 11

Exceptions

Exceptions are runtime errors, which may be handled gracefully by F#. Exceptions are handled by the "try" keyword both in expressions. E.g., Integer division by zero raises and exception, but it may be handled in a script as follows,

Listing 11.1, exceptionDivByZero.fsx:
A division by zero is caught and a default value is returned.

```
let div enum denom =
   try
      enum / denom
   with
      | :? System.DivideByZeroException -> System.Int32.MaxValue

printfn "3 / 1 = %d" (div 3 1)
printfn "3 / 0 = %d" (div 3 0)
```

```
3 / 1 = 3
3 / 0 = 2147483647
```

The "try" expressions have the following syntax,

Listing 11.2:

```
expr = ...
  | "try" expr "with" ["|"] rules (*exception*)
  | "try" expr "finally" expr; (*exception with cleanup*)

rules = rule | rule "|" rules;
rule = pat ["when" expr] "->" expr;
```

Exceptions are a basic-type called exn, and F# has a number of built-in, see Table 11.1. [1] The programs may define new exceptions using the syntax,

[1] Todo: **Add example of raising builtin exceptions, e.g.,** `let f () : int = raise (System.ArithmeticException "hej") in f();;`

Attribute	Description
`System.ArithmeticException`	Failed arithmetic operation.
`System.ArrayTypeMismatchException`	Failed attempt to store an element in an array failed because of type mismatch.
`System.DivideByZeroException`	Failed due to division by zero.
`System.IndexOutOfRangeException`	Failed to access an element in an array because the index is less than zero or equal or greater than the length of the array.
`System.InvalidCastException`	Failed to explicitly convert a base type or interface to a derived type at run time.
`System.NullReferenceException`	Failed use of a `null` reference was used, since it required the referenced object.
`System.OutOfMemoryException`	Failed to use `new` to allocate memory.
`System.OverflowException`	Failed arithmetic operation in a checked context which caused an overflow.
`System.StackOverflowException`	Failed use of the internal stack caused by too many pending method calls, e.g., from deep or unbounded recursion.
`System.TypeInitializationException`	Failed initialization of code for a type, which was not caught.

Table 11.1: Built-in exceptions.

Listing 11.3:

```
"exception" ident of typeTuple (*exception definition*)
typeTuple = type | type "*" typeTuple;
```

and any exceptions may be *raised* using the functions "`failwith`", "`invalidArg`", "`raise`", and "`reraise`". An example of raising an exception with the `raise` function is,

· raise an exception

Listing 11.4, exceptionDefinition.fsx:
A user-defined exception is raised but not caught by outer construct.

```
exception DontLikeFive of string

let picky a =
  if a = 5 then
    raise (DontLikeFive "5 sucks")
  else
    a

printfn "picky %A = %A" 3 (picky 3)
printfn "picky %A = %A" 5 (picky 5)
```

```
picky 3 = 3
FSI_0001+DontLikeFive: Exception of type 'FSI_0001+DontLikeFive' was
    thrown.
  at FSI_0001.picky (Int32 a) <0x66f3f58 + 0x00057> in <filename unknown
    >:0
  at <StartupCode$FSI_0001>.$FSI_0001.main@ () <0x66f31a0 + 0x0017f> in <
    filename unknown>:0
  at (wrapper managed-to-native) System.Reflection.MonoMethod:
    InternalInvoke (System.Reflection.MonoMethod,object,object[],System.
    Exception&)
  at System.Reflection.MonoMethod.Invoke (System.Object obj, BindingFlags
    invokeAttr, System.Reflection.Binder binder, System.Object[]
    parameters, System.Globalization.CultureInfo culture) <0x1a7c270 + 0
    x000a1> in <filename unknown>:0
Stopped due to error
```

Here an exception called `DontLikeFive` is defined, and it is raised in the function `picky`. When run, F# stops at run-time after the program has raised the exception with a long description of the reason including the name of the exception. Exceptions include messages, and the message for `DontLikeFive` is of type `string`. This message is passed to the "`try`" expression and may be processed as e.g.,

Listing 11.5, exceptionDefinitionNCatch.fsx:
Catching a user-defined exception.

```
exception DontLikeFive of string

let picky a =
  if a = 5 then
    raise (DontLikeFive "5 sucks")
  else
    a

try
  printfn "picky %A = %A" 3 (picky 3)
  printfn "picky %A = %A" 5 (picky 5)
with
  | DontLikeFive msg -> printfn "Exception caught with message: %s" msg
```

```
picky 3 = 3
Exception caught with message: 5 sucks
```

Note that the type of `picky` is `a:int -> int` because its argument is compared with an integer in the conditional statement. This contradicts the typical requirements for "`if`" statements, where every branch has to return the same type. However, any code that explicitly raises exceptions are ignored, and the type is inferred by the remaining branches.

The `failwith : string -> exn` function takes a string and raises the built-in `System.Exception` exception,

Listing 11.6, exceptionFailwith.fsx:
An exception raised by `failwith`.

```
if true then failwith "hej"
```

```
System.Exception: hej
  at <StartupCode$FSI_0001>.$FSI_0001.main@ () <0x676f158 + 0x00037> in <
    filename unknown>:0
  at (wrapper managed-to-native) System.Reflection.MonoMethod:
    InternalInvoke (System.Reflection.MonoMethod,object,object[],System.
    Exception&)
  at System.Reflection.MonoMethod.Invoke (System.Object obj, BindingFlags
    invokeAttr, System.Reflection.Binder binder, System.Object[]
    parameters, System.Globalization.CultureInfo culture) <0x1a7c270 + 0
    x000a1> in <filename unknown>:0
Stopped due to error
```

To catch the `failwith` exception, there are two choices, either use the `:?` or the `Failure` pattern. the `:?` pattern matches types, and we can match with the type of `System.Exception` as,

Listing 11.7, exceptionSystemException.fsx:
Catching a `failwith` exception using type matching pattern.

```
let _ =
  try
    failwith "Arrrrg"
  with
    :? System.Exception -> printfn "So failed"
```

```
/Users/sporring/repositories/fsharpNotes/src/exceptionSystemException.fsx
    (5,5): warning FS0067: This type test or downcast will always hold


/Users/sporring/repositories/fsharpNotes/src/exceptionSystemException.fsx
    (5,5): warning FS0067: This type test or downcast will always hold
So failed
```

However, this gives annoying warnings, since F# internally is built such that all exception matches the type of `System.Exception`. Instead it is better to either match anything,

Listing 11.8, exceptionMatchWildcard.fsx:
Catching a `failwith` exception using the wildcard pattern.

```
let _ =
  try
    failwith "Arrrrg"
  with
    _ -> printfn "So failed"
```

```
So failed
```

or use the built-in `Failure` pattern,

Listing 11.9, exceptionFailure.fsx:
Catching a `failwith` exception using the `Failure` pattern.

```
let _ =
  try
    failwith "Arrrrg"
  with
    Failure msg ->
      printfn "The castle of %A" msg
```

```
The castle of "Arrrrg"
```

Notice how only the `Failure` pattern allows for the parsing of the message given to `failwith` as argument.

The `invalidArg` takes 2 strings and raises the built-in `ArgumentException`

**Listing 11.10, exceptionInvalidArg.fsx:
An exception raised by `invalidArg`.**

```
if true then invalidArg "a" "is too much 'a'"
```

```
System.ArgumentException: is too much 'a'
Parameter name: a
  at <StartupCode$FSI_0001>.$FSI_0001.main@ () <0x666f1f0 + 0x0005b> in <
    filename unknown>:0
  at (wrapper managed-to-native) System.Reflection.MonoMethod:
    InternalInvoke (System.Reflection.MonoMethod,object,object[],System.
    Exception&)
  at System.Reflection.MonoMethod.Invoke (System.Object obj, BindingFlags
    invokeAttr, System.Reflection.Binder binder, System.Object[]
    parameters, System.Globalization.CultureInfo culture) <0x1a7c270 + 0
    x000a1> in <filename unknown>:0
Stopped due to error
```

This would be caught by type matching as,

**Listing 11.11, exceptionInvalidArgNCatch.fsx:
Catching the exception raised by `invalidArg`.**

```
let _ =
  try
    invalidArg "a" "is too much 'a'"
  with
    :? System.ArgumentException -> printfn "Argument is no good!"
```

```
Argument is no good!
```

The "`try`" construction is typically used to gracefully handle exceptions, but there are times, where you may want to pass on the bucket, so to speak, and reraise the exception. This can be done with the "`reraise`".

Listing 11.12, exceptionReraise.fsx:
Reraising an exception.

```
let _ =
  try
    failwith "Arrrrg"
  with
    Failure msg ->
      printfn "The castle of %A" msg
      reraise()
```

```
The castle of "Arrrrg"
System.Exception: Arrrrg
  at <StartupCode$FSI_0001>.$FSI_0001.main@ () <0x6745e88 + 0x00053> in <
    filename unknown>:0
Stopped due to error
```

The `reraise` function is only allowed to be the final call in the expression of a "`with`" rule.

At exceptions, it is not always obvious what should be returned. E.g., in the Listing 11.1, the exception is handled gracefully, but the return value is somewhat arbitrarily chosen to be the largest possible integer, which is still far from infinity, which is the correct result. Instead we could use the *option type*. · option type
The option type is a wrapper, that can be put around any type, and which extends the type with the special value `None`. All other values are preceded by the `Some` identifier. E.g., to rewrite Listing 11.1 to correctly represent the non-computable value, we could write

Listing 11.13: Option types can be used, when the value in case of exceptions is unclear.

```
> let div enum denom =
-   try
-     Some (enum / denom)
-   with
-     | :? System.DivideByZeroException -> None;;

val div : enum:int -> denom:int -> int option

>
- let a = div 3 1;;

val a : int option = Some 3

> let b = div 3 0;;

val b : int option = None
```

The value of an option type can be extracted by and tested for by its member function, `IsNone`, `IsSome`, and `Value`, e.g.,

Listing 11.14, option.fsx:
Simple operations on option types.

```
let a = Some 3;
let b = None;
printfn "%A %A" a b
printfn "%A %b %b" a.Value b.IsSome b.IsNone
```

```
Some 3 <null>
3 false true
```

In the “`try`”-“`finally`”, the “`finally`” expression is always executed, e.g.,

Listing 11.15, exceptionFinally.fsx:
The “`finally`” expression in “`try`”-“`finally`” will always be executed.

```
let _ =
  try
    if true then failwith "True"
    else failwith "False"
  finally
    printfn "Finally expression will always be executed."
```

```
Finally expression will always be executed.
System.Exception: True
  at <StartupCode$FSI_0001>.$FSI_0001.main@ () <0x6745328 + 0x0003f> in <
    filename unknown>:0
  at (wrapper managed-to-native) System.Reflection.MonoMethod:
    InternalInvoke (System.Reflection.MonoMethod,object,object[],System.
    Exception&)
  at System.Reflection.MonoMethod.Invoke (System.Object obj, BindingFlags
    invokeAttr, System.Reflection.Binder binder, System.Object[]
    parameters, System.Globalization.CultureInfo culture) <0x1a7c270 + 0
    x000a1> in <filename unknown>:0
Stopped due to error
```

This is useful for cleaning up, e.g., closing files etc. which we will discuss in Chapter 12. The only way to combine “`try`”-“`finally`” with “`try`”-“`with`” is to nest the expression inside each other.

[2]

[2]Todo: **Husk, som if-else skal try-with branches være af samme type. Giv eksempel på parsning af argument til selvdefineret exception. Gå problem med divmed0exception igennem: 3 cases, failsafe værdier, undtagelser, eller option typer.**

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 built-in namespaces and classes. The `printf` family of functions is defined in the `Printf` module of the `Fsharp.Core'` namespace, and it was discussed in Chapter 6.4, 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 a UTF8 string using `fprintf` in order to store them to file in a human readable form, and interpreted from UTF8 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, making notes and thus saving 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. In contrast, files are finite in size and allow for global operations on all the file's data. Files may be considered a stream, but the opposite is not true. · stream

12.1 Interacting with the console

[1] From a programming perspective, the console is a stream: A 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, nor can we peak into the future and retrieve what the user will input in the future. The console uses 3 built-in streams in `System.Console`,,,

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.

[2] 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: string -> unit`	Write to the console. E.g., `System.Console.Write "Hello world."`.
`WriteLine: string -> unit`	As `Write` but followed by a newline character, e.g., `System.Console.WriteLine "Hello world."`.
`Read: unit -> int`	Read the next key from the keyboard blocking execution as long, e.g., `System.Console.Read ()`.
`ReadKey: unit -> System.ConsoleKeyInfo`	As `Read` but writing the key to the console as well, e.g. , `System.Console.ReadKey ()`.
`ReadLine unit -> string`	Read the next sequence of characters until newline from the keyboard, e.g. , `System.Console.ReadLine ()`.

[3] Note that you must supply the empty argument "`()`" to the `Read` functions, in order to run most of the functions instead of referring to them as values. The `System.ConsoleKeyInfo` object contains the key pressed as the `KeyChar` member as well as other information about the event. A short demonstration script is given in Listing 12.1.

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

```
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))
```

An example dialogue using Listing 12.1 is,

[1]Todo: **Spec-4.0 Section 18.2.9**
[2]Todo: **Tilføj** `System.Console.Error.WriteLine(``Goodbye, World!'');`
[3]Todo: **Tilføj eksemple på ConsoleModifiers, f.eks.** `readKey.fsx`

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

Thus, `Write` and `WriteLine` acts as `printfn` but without a formatting string. **For writing to the console, `printf` is to be preferred.** Advice

12.2 Storing and retrieving data from a file

A file stored on the filesystem has a name, and it must be opened before it can be accessed and closed when finished. Opening files informs the operating system that your program is now going to use the file, and your program may request protection of the file from the operating system. E.g., if you are going to write to the file, then this typically implies that no one else may write to the file at the same time, since simultaneous writing to a file may leave the resulting file in an uncertain state. Thus, you reserve a file by opening it, and you release it again by closing it. Sometimes the operating system will realize that a file, that was opened by a program, is no longer being used, e.g., since the program is no longer running, but **it is good practice always to release reserved files, e.g., by closing them as soon as possible, such that other programs may have access to it.** Advice On the other hand, it is typically safe for several programs to read the same file at the same time, but it is still important to close files after their use, such that the operating system can effectively manage the computer's resources. Reserved files are just one of the possible obstacles that you may meet when attempting to open a file. Other points of failure may be that the file may not exist, your program may not have sufficient rights for accessing it, or the device, where the file is stored, may have unreliable access. Thus, **never assume that accessing files always works, but program defensively, e.g., by checking the return status of the file accessing functions and by "`try`" constructions.** Advice

Data in a files may have been stored in various ways, e.g., it may contain UTF8 encoded characters or sequences of floating point numbers stored as raw bits in chunks of 64 bits, or it may be a sequence of bytes that are later going to be interpreted as an image in jpeg or tiff format. To aid in retrieving the data, F# has a family of open functions, all residing in the `System.IO.File` class. These are described in Table 12.1. For the general `Open` function, you must also specify how the file is to be opened. This is done with a special set of values described in Table 12.2. An example of how a file is opened and later closed is shown in Listing 12.2.

Listing 12.2, openFile.fsx:
Opening and closing a file, in this case the source code of this same file.

```
let filename = "openFile.fsx"

let reader =
  try
    Some (System.IO.File.Open (filename, System.IO.FileMode.Open))
  with
    _ -> None

if reader.IsSome then
  printfn "The file %A was successfully opened." filename
  reader.Value.Close ()
```

```
The file "openFile.fsx" was successfully opened.
```

System.IO.File	Description
Open: (path : string) * (mode : FileMode) -> FileStream	Request the opening of a file on path for reading and writing with access mode FileMode, see Table 12.2. Other programs are not allowed to access the file, before this program closes it.
OpenRead: (path : string) -> FileStream	Request the opening of a file on path for reading. Other programs may read the file regardless of this opening.
OpenText: (path : string) -> StreamReader	Request the opening of an existing UTF8 file on path for reading. Other programs may read the file regardless of this opening.
OpenWrite: (path : string) -> FileStream	Request the opening of a file on path for writing with FileMode.OpenOrCreate. Other programs may not access the file, before this program closes it.
Create: (path : string) -> FileStream	Request the creation of a file on path for reading and writing, overwriting any existing file. Other programs may not access the file, before this program closes it.
CreateText: (path : string) -> StreamWriter	Request the creation of an UTF8 file on path for reading and writing, overwriting any existing file. Other programs may not access the file, before this program closes it.

Table 12.1: The family of System.IO.File.Open functions. See Table 12.2, 12.3, 12.4, ??, 12.5, ??, and 12.6 for the description of FileMode, FileStream, StreamWriter, and StreamReader.

FileMode.	Description
Append	Open a file and seek to its end, if it exists, or create a new file. Can only be used together with FileAccess.Write. May throw IOException and NotSupportedException exceptions.
Create	Create a new file, and delete an already existing file. May throw the UnauthorizedAccessException exception.
CreateNew	Create a new file, but throw the IOException exception, if the file already exists.
Open	Open an existing file, and System.IO.FileNotFoundException exception is thrown if the file does not exist.
OpenOrCreate	Open a file, if exists, or create a new file.
Truncate	Open an existing file and truncate its length to zero. Cannot be used together with FileAccess.Read.

Table 12.2: File mode values for the System.IO.Open function.

Property	Description
`CanRead`	Gets a value indicating whether the current stream supports reading.(Overrides Stream.CanRead.)
`CanSeek`	Gets a value indicating whether the current stream supports seeking.(Overrides Stream.CanSeek.)
`CanWrite`	Gets a value indicating whether the current stream supports writing.(Overrides Stream.CanWrite.)
`Length`	Gets the length in bytes of the stream.(Overrides Stream.Length.)
`Name`	Gets the name of the FileStream that was passed to the constructor.
`Position`	Gets or sets the current position of this stream.(Overrides Stream.Position.)

Table 12.3: Some properties of the `System.IO.FileStream` class.

`Method`	Description
`Close ()`	Closes the stream.
`Flush ()`	Causes any buffered data to be written to the file.
`Read byte[] * int * int`	Reads a block of bytes from the stream and writes the data in a given buffer.
`ReadByte ()`	Read a byte from the file and advances the read position to the next byte.
`Seek int * SeekOrigin`	Sets the current position of this stream to the given value.
`Write byte[] * int * int`	Writes a block of bytes to the file stream.
`WriteByte byte`	Writes a byte to the current position in the file stream.

Table 12.4: Some methods of the `System.IO.FileStream` class.

Notice how the example uses the defensive programming style, where the "`try`"-expression is used to return the optional datatype, and further processing is made dependent on the success of the opening operation.

In Fsharp the distinction between files and streams are not very clear. Fsharp offers built-in support for accessing files as bytes through the `System.IO.FileStream` class, and for characters in a particular encoding through the `System.IO.TextReader` and `System.IO.TextWriter`.

A successfully opened `System.IO.FileStream` file, e.g., using `System.IO.File.OpenRead` from Table 12.1, will result in an `FileStream` object. From this object we can extract information about the file, such as the permitted operations and more listed in Table 12.3. This information is important in order to restrict the operation that we will perform on the file. Some typical operations are listed in and 12.4. E.g., we may `Seek` a particular position in the file, but only within the range of legal postions from 0 until the length of the file. Most operating systems do not necessarily write information to files immediately after one of the `Write` functions, but will often for optimization purposes will often collect information in a buffer that is to be written to a device in batches. However, sometimes is is useful to be able to force the operating system to empty its buffer to the device. This is called *flushing* · flushing
and can be forced using the `Flush` function.

Text is typically streamed through the `StreamReader` and `StreamWriter`. These may be considered higher order stream processing, since they include an added interpretation of the bits to strings. A `StreamReader` has methods similar to a `FileStream` object and a few new properties and methods, such as the `EndOfStream` property and `ReadToEnd` method, see Table 12.5. Likewise, a `StreamWriter` has an added method for automatically flushing following every writing operation. A simple example of opening a text-file and processing it is given in Listing 12.3.

Property/Method	Description
`EndOfStream`	Check whether the stream is at its end.
`Close ()`	Closes the stream.
`Flush ()`	Causes any buffered data to be written to the file.
`Peek ()`	Reads the next character, but does not advance the position.
`Read ()`	Reads the next character.
`Read char[] * int * int`	Reads a block of bytes from the stream and writes the data in a given buffer.
`ReadLine ()`	Reads the next line of characters until a newline. Newline is discarded.
`ReadToEnd ()`	Reads the remaining characters till end-of-file.

Table 12.5: Some methods of the `System.IO.StreamReader` class.

Property/Method	Description
`AutoFlush : bool`	Get or set the auto-flush. If set, then every call to `Write` will flush the stream.
`Close ()`	Closes the stream.
`Flush ()`	Causes any buffered data to be written to the file.
`Write 'a`	Write a basic type to the file.
`WriteLine string`	As `Write` but followed by newline.

Table 12.6: Some methods of the `System.IO.StreamWriter` class.

Listing 12.3, readFile.fsx:
An example of opening a text file, and using the `StreamReader` properties and methods.

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

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

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

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

Here the program reads the source code of itself, and prints it to the console.

12.3 Working with files and directories.

Fsharp has support for managing files summarized in the `System.IO.File` class and summarized in Table 12.7

Function	Description
`Copy (src : string, dest : string)`	Copy a file from `src` to `dest` possibly overwriting any existing file.
`Delete string`	Delete a file
`Exists string`	Check whether the file exists
`Move (from : string, to : string)`	Move a file from `src` to `to` possibly overwriting any existing file.

Table 12.7: Some methods of the `System.IO.File` class.

Function	Description
`CreateDirectory string`	Create the directory and all implied sub-directories.
`Delete string`	Delete a directory
`Exists string`	Check whether the directory exists
`GetCurrentDirectory ()`	Get working directory of the program
`GetDirectories (path : string)`	Get directories in `path`
`GetFiles (path : string)`	Get files in `path`
`Move (from : string, to : string)`	Move a directory and its content from `src` to `to`.

Table 12.8: Some methods of the `System.IO.Directory` class.

In the `System.IO.Directory` class there are a number of other frequently used functions summarized in Table 12.8.

In the `System.IO.Path` class there are a number of other frequently used functions summarized in Table 12.9.

12.4 Reading from the internet

The internet is a global collection of computers that are connected in a network using the internet protocol suite TCP/IP. The internet is commonly used for transport of data such as emails and for offering services such as web pages on the World Wide Web. Web resources are identified by a *Uniform Resource Locator* (*URL*) popularly known as a web page, and an URL contains information about where and how data from the web page is to be obtained. E.g., the URL `https://en.wikipedia.org/wiki/F_Sharp_(programming_language)`, contains 3 pieces of information: `https` is the protocol to

· Uniform Resource Locator
· URL

Function	Description
`Combine string * string`	Combine 2 paths into a new path.
`GetDirectoryName (path: string)`	Extract the directory name from `path`.
`GetExtension (path: string)`	Extract the extension from `path`.
`GetFileName (path: string)`	Extract the name and extension from `path`.
`GetFileNameWithoutExtension (path : string)`	Extract the name without the extension from `path`.
`GetFullPath (path : string)`	Extract the absolute path from `path`.
`GetTempFileName ()`	Create a uniquely named and empty file on disk and return its full path.

Table 12.9: Some methods of the `System.IO.Path` class.

be used to interact with the resource, `en.wikipedia.org` is the host's name, and `wiki/F_Sharp_(programming_language)` is the filename.

Fsharp's `System` namespace contains functions for accessing web pages as stream as illustrated in Listing 12.4.

Listing 12.4, webRequest.fsx:
Downloading a web page and printing the first few characters.

```
/// Set a url up as a stream
let url2Stream url =
    let uri = System.Uri url
    let request = System.Net.WebRequest.Create uri
    let response = request.GetResponse ()
    response.GetResponseStream ()

/// Read all contents of a web page as a string
let readUrl url =
    let stream = url2Stream url
    let reader = new System.IO.StreamReader(stream)
    reader.ReadToEnd ()

let url = "http://fsharp.org"
let a = 40

let html = readUrl url
printfn "Downloaded %A. First %d characters are: %A" url a html.[0..a]
```

```
Downloaded "http://fsharp.org". First 40 characters are: "<!DOCTYPE html>
<html lang="en">
<head>"
```

To connect to a URL as a stream, we first need first format the URL string as a *Uniform Resource Identifiers* (*URI*), which is a generalization of the URL concept, using the `System.Uri` function. Then we must initialize the request by the `System.Net.WebRequest` function, and the response from the host is obtained by the `GetResponse` method. Finally, we can access the response as a stream by the `GetResponseStream` method. In the end, we convert the stream to a `StreamReader`, such that we can use the methods from Table 12.5 to access the web page.

· Uniform Resource Identifiers
· URI

[4]

12.5 Programming intermezzo

A typical problem, when working with files, is

Problem 12.1:

Ask the user for the name of an existing file.

[4]Todo: **Add section on command line arguments, e.g., commandLineArgs.fsx and get CommandLineArgs.fsx. Note difference between running these as compiled and interpreted code.**

Such a dialogue most often requires the program to aid the user, e.g., by telling the user, which files are available, and to check that the filename entered is an existing file. A solution could be,

Listing 12.5, filenamedialogue.fsx:

```
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
```

A practice problem could be,

Problem 12.2:
Ask the user for the name of an existing file, read the file and print it in reverse order.

This could be solved as,

Listing 12.6, reverseFile.fsx:

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

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"]
```

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