Learning to Program with F#

Jon Sporring

Department of Computer Science,
University of Copenhagen

2018-11-12 13:54:23+01:00

Contents

16 Higher Order Functions

A *higher order function* is a function that takes a function as an argument and/or returns a function. Higher order functions are also sometimes called functionals or functors. F# is a functions-first programming language with strong support for working with functions as values: Functions evaluate as *closures*, see Section 6.2, which can be passed to and from functions as any other value. An example of a higher order function is `List.map` which takes a function and a list and produces a list, demonstrated in Listing 16.1.

· higher order function

· closures

Listing 16.1 higherOrderMap.fsx:
`List.map` is a higher order function, since it takes a function as argument.

```
let inc x = x + 1
let newList = List.map inc [2; 3; 5]
printfn "%A" newList
```

```
$ fsharpc --nologo higherOrderMap.fsx && mono
  higherOrderMap.exe
[3; 4; 6]
```

Here `List.map` applies the function `inc` to every element of the list. Higher order functions are often used together with *anonymous functions*, where the anonymous functions is given as argument. For example, Listing 16.1 may be rewritten using an anonymous function as shown in Listing 16.2.

· anonymous functions

Listing 16.2 higherOrderAnonymous.fsx:
An anonymous function is a higher order function used here as an unnamed argument. Compare with Listing 16.1.

```
let newList = List.map (fun x -> x + 1) [2; 3; 5]
printfn "%A" newList
```

```
$ fsharpc --nologo higherOrderAnonymous.fsx
$ mono higherOrderAnonymous.exe
[3; 4; 6]
```

The code may be compacted even further, as shown in Listing 16.3.

Listing 16.3 higherOrderAnonymousBrief.fsx:
A compact version of Listing 16.1.

```
printfn "%A" (List.map (fun x -> x + 1) [2; 3; 5])
```

```
$ fsharpc --nologo higherOrderAnonymousBrief.fsx
$ mono higherOrderAnonymousBrief.exe
[3; 4; 6]
```

What was originally three lines in Listing 16.1 including bindings to the names `inc` and `newList` has in Listing 16.3 been reduced to a single line with no bindings. All three programs result in the same output and as such are equal. Likewise, running times will be equal. However, they differ in readability for a programmer and ease of bug hunting and future maintenance: Bindings allows us to reuse the code at a later stage, but if there is no reuse, then the additional bindings may result in a cluttered program. Further, for compact programs like Listing 16.3, it is not possible to perform a unit test of the function arguments. Finally, bindings emphasize semantic aspects of the evaluation being performed merely by the names we select, and typically long, meaningful names are to be preferred, within reasonable limits. For example instead of `inc` one could have used `increment_by_one` or similar which certainly is semantically meaningful, but many programmers will find that the short is to be preferred in order to reduce the amount of typing to be performed.

Anonymous functions are also useful as return values of functions, as shown in Listing 16.4

Listing 16.4 higherOrderReturn.fsx:
The procedure `inc` returns an increment function. Compare with Listing 16.1.

```
let inc n =
  fun x -> x + n
printfn "%A" (List.map (inc 1) [2; 3; 5])
```

```
$ fsharpc --nologo higherOrderReturn.fsx && mono
   higherOrderReturn.exe
[3; 4; 6]
```

Here the `inc` function produces a customized incrementation function as argument to `List.map`: It adds a prespecified number to an integer argument. Note that the closure of this customized function is only produced once, when the arguments for `List.map` is prepared, and not every time `List.map` maps the function to the elements of the list. Compare with Listing 16.1.

Piping is another example of a set of higher order function: `(<|)`, `(|>)`, `(<||)`, `(||>)`, `(<|||)`, `(|||>)`.[1] E.g., the functional equivalent of the right-to-left piping operator takes a value and a function and applies the function to the value, as demonstrated in Listing 16.5. · piping

[1] Jon: **Make piping operators go into index.**

Listing 16.5 higherOrderPiping.fsx:
The functional equivalent of the right-to-left piping operator is a higher order function.

```
let inc x = x + 1
let aValue = 2
let anotherValue = (|>) aValue inc
printfn "%d -> %d" aValue anotherValue
```

```
$ fsharpc --nologo higherOrderPiping.fsx && mono
  higherOrderPiping.exe
2 -> 3
```

Here the piping operator is used to apply the `inc` function to `aValue`. A more elegant way to write this would be `aValue |> inc`, or even just `inc aValue`.

16.1 Function Composition

Piping is a useful shorthand for composing functions, where the focus is on the transformation of arguments and results. Using higher order functions, we can forgo the arguments and compose functions as functions directly. This is done with the "`>>`" and "`<<`" operators. An example is given in Listing 16.6.

· function composition
· >>
· <<

Listing 16.6 higherOrderComposition.fsx:
Functions defined as compositions of other functions.

```
let f x = x + 1
let g x = x * x
let h = f >> g
let k = f << g
printfn "%d" (g (f 2))
printfn "%d" (h 2)
printfn "%d" (f (g 2))
printfn "%d" (k 2)
```

```
$ fsharpc --nologo higherOrderComposition.fsx
$ mono higherOrderComposition.exe
9
9
5
5
```

In the example we see that `(f >> g) x` gives the same result as `g (f x)`, while `(f << g) x` gives the same result as `f (g x)`. A memo technique for remembering the order of the application, when using the function composition operators, is that `(f >> g) x` is the same as `x |> f |> g`, i.e., the result of applying `f` to `x` is the argument to `g`. However, there is a clear distinction between the piping and composition operators. The type of the piping operator is

```
(|>) : ('a, 'a -> 'b) -> 'b
```

i.e., the piping operator takes a value of type `'a` and a function of type `'a -> 'b`, applies the function to the value, and produces the value `'b`. In contrast, the composition operator has type

```
(>>) : ('a -> 'b, 'b -> 'c) -> ('a -> 'c)
```

i.e., it takes two functions of type `'a -> 'b` and `'b -> 'c` respectively, and produces a new function of type `a' -> 'c`.

16.2 Currying

Consider a function `f` of two generic arguments. Its type in F# will be `f : 'a -> 'b -> 'c`, meaning that `f` takes an argument of type `'a` and returns a function of type `'b -> 'c`. That is, if just one argument is given, then the result is a function, not a value. This is called *partial specification* or *currying* in tribute of Haskell Curry[2]. An example is given in Listing 16.7.

· partial specification
· currying

Listing 16.7 higherOrderCurrying.fsx:
Currying: defining a function as a partial specification of another.

```
let mul x y = x*y
let timesTwo = mul 2.0
printfn "%g" (mul 5.0 3.0)
printfn "%g" (timesTwo 3.0)
```

```
$ fsharpc --nologo higherOrderCurrying.fsx
$ mono higherOrderCurrying.exe
15
6
```

Here, `mul 2.0` is a partial application of the function `mul x y`, where the first argument is fixed, and hence `timesTwo` is a function of 1 argument being the second argument of `mul`. The same can be achieved using tuple arguments, as shown in Listing 16.8.

Listing 16.8 higherOrderTuples.fsx:
Partial specification of functions using tuples is less elegant. Compare with Listing 16.7.

```
let mul (x, y) = x*y
let timesTwo y = mul (2.0, y)
printfn "%g" (mul (5.0, 3.0))
printfn "%g" (timesTwo 3.0)
```

```
$ fsharpc --nologo higherOrderTuples.fsx && mono
  higherOrderTuples.exe
15
6
```

[2] Haskell Curry (1900–1982) was an American mathematician and logician who also has a programming language named after him: Haskell.

Conversion between multiple and tuple arguments is easily done with higher order functions, as demonstrated in Listing 16.9.

Listing 16.9: Two functions to convert between two and 2-tuple arguments.

```
> let curry f x y = f (x,y)
- let uncurry f (x,y) = f x y;;
val curry : f:('a * 'b -> 'c) -> x:'a -> y:'b -> 'c
val uncurry : f:('a -> 'b -> 'c) -> x:'a * y:'b -> 'c
```

Conversion between multiple and tuple arguments are useful when working with higher order functions such as `Lst.map`. E.g., if `let mul (x, y) = x * y` as in Listing 16.8, then `curry mul` has the type `x:'a -> y:'b -> 'c` as can be seen in Listing 16.9, and thus is equal to the anonymous function `fun x y -> x * y`. Hence, `curry mul 2.0` is equal to `fun y -> 2.0 * y`, since the precedence of function calls is `(curry mul) 2.0`.

Currying makes elegant programs and is often used in functional programming. Nevertheless, currying may lead to obfuscation, and in general, **currying should be used with care and be well documented for proper readability of code.** Advice

17 | The functional programming paradigm

Functional programming is a style of programming which performs computations by evaluating functions. Functional programming avoids mutable values and side-effects. It is declarative in nature, e.g., by the use of value- and function-bindings – `let`-bindings – and avoids statements – `do`-bindings. Thus, the result of a function in functional programming depends only on its arguments, and therefore functions have no side-effect and are deterministic, such that repeated call to a function with the same arguments always gives the same result. In functional programming, data and functions are clearly separated, and hence data structures are dum as compared to objects in object-oriented programming paradigm, see Chapter 22. Functional programs clearly separate behavior from data and subscribes to the view that *it is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.* Simplifying the data structure has the advantage that it is much easier to communicate data than functions and procedures between programs and environments. The .Net, mono, and java's virtual machine are all examples of an attempt to rectify this, however, the argument still holds.

The functional programming paradigm can trace its roots to lambda calculus introduced by Alonzo Church in 1936 [1]. Church designed lambda calculus to discuss computability. Some of the forces of the functional programming paradigm are that it is often easier to prove the correctness of code, and since no states are involved, then functional programs are often also much easier to parallelize than other paradigms.

Functional programming has a number of features:

· pure function

Pure functions

Functional programming is performed with pure functions. A pure function always returns the same value, when given the same arguments, and it has no side-effects. A function in F# is an example of a pure function. Pure functions can be replaced by their result without changing the meaning of the program. This is known as *referential transparency*.

· referential transparency

Higher order functions

· higher order function

Functional programming makes use of higher order functions, where functions may be given as arguments and returned as results of a function application. Higher order functions and *first-class citizenship* are related concepts, where higher-order functions are the mathematical description of functions that operator on functions, while a first-class citizen is the computer science term for functions as values. F# implements higher-order functions.

· first-class citizenship

· recursion

Recursion

Functional programs use recursion instead of `for`- and `while`-loops. Recursion can make programs ineffective, but compilers are often designed to optimize tail-recursion calls. Common recursive programming structures are often available as optimized higher-order functions such as *iter*, *map*, *reduce*, *fold*, and *foldback*. F# has good support for all of these features.

· iter
· map
· reduce
· fold
· foldback
· immutable state
· immutable state
· strongly typed

Immutable states
Functional programs operate on values not on variables. This implies lexicographical scope in contrast to mutable values, which implies dynamic scope.

Strongly typed
Functional programs are often strongly typed, meaning that types are set no later than at compile-time. F# does have the ability to perform runtime type assertion, but for most parts it relies on explicit type annotations and type inference at compile-time. The implication is that type errors are caught at compile time instead of at runtime.

· lazy evaluation

Lazy evaluation
Due to referential transparency, values can be compute any time up until the point, when needed. Hence, they need not be computed at compilation time, which allows for infinite data structures. F# has support for lazy evaluations using the `lazy`-keyword, sequences using the `seq`-type, and computation expressions, all of which are advanced topics and not treated in this book.

· `lazy`
· `seq`

Immutable states imply that data structures in functional programming are different than in imperative programming. E.g., in F# lists are immutable, so if an element of a list is to be changed, a new list must be created by copying all old values except that which is to be changed. Such an operation is therefore linear in computational complexity. In contrast, arrays are mutable values, and changing a value is done by reference to the value's position and change the value at that location. This has constant computational complexity. While fast, mutable values give dynamic scope, makes reasoning about the correctness of a program harder, since mutable states do not have referential transparency.

Functional programming may be considered a subset of *imperative programming*, in the sense that functional programming does not include the concept of a state, or one may think of functional programming as only having one unchanging state. Functional programming has also a bigger focus on declaring rules for *what* should be solved, and not explicitly listing statements describing *how* these rules should be combined and executed in order to solve a given problem. Functional programming is often found to be less error-prone at runtime making more stable, safer programs, and less open for, e.g., hacking.

· imperative programming

17.1 Functional design

A key to all good programming designs is encapsulating code into modules. For functional programs, the essence is to consider data and functions as transformations of data. I.e., the basic pattern is a piping sequence,

`x |> f |> g |> h`,

where `x` is the input data, and `f`, `g`, and `h` are functions that transform the data. Of course, most long programs include lots of control structure implying that we would need junctions in the pipe system, however, piping is a useful memo technique.

In functional programming there are some pitfalls, that you should avoid:

- Creating large data structures, such as a single record containing all data. Since data is immutable, changing a single field in a monstrous record would mean a lot of copying in many parts of your program. Better is to use a range of data structures that express isolated semantic units of your problem.

- Non-tail recursion. Relying on the built-in functions `map`, `fold`, etc., is a good start for efficiency.

- Single character identifiers. Since functional programming tends to produce small, well-defined functions, there is a tendency to the use of single character identifiers, e.g., `let f x = ...`. In the very small, this can be defended, but the names used as identifiers can be used to increase the readability of code to yourself or to others. Typically, identifiers are long and informative in the outer most scope, while decreasing in size as you move in.

- Few comments. Since functional programming is very concise, there is a tendency, that we as programmers forget to add sufficient comments to the code, since at the time of writing, the meaning may be very clear and well thought through, but experience shows that this clarity deteriorates fast with time.

- Identifiers that are meaningless clones of each other. Since identifiers cannot be reused except by overshadowing in deeper scopes, there is often a tendency to have a family of identifiers like `a`, `a2`, `newA` etc. Better is to use names, that more clearly states the semantic meaning of the values, or, if only used as temporary storage, to discard them completely in lieu of piping and function composition. However, the latter most often requires comments describing the transformation being performed.

Thus, a design pattern for functional programs must focus on,

- What is the input data to be processed

- How is the data to be transformed

For large programs, the design principle is often similar to other programming paradigms, which are often visualized graphically as components that take input, interact, and produces results often together with a user. The effect of functional programming is mostly seen in the small, i.e., where a subtask is to be structure functionally.

18 | Handling Errors and Exceptions

18.1 Exceptions

Exceptions are runtime errors, such as division by zero. E.g., attempting integer division by zero halts execution and a long somewhat cryptic error message is written to screen as illustrated in Listing 18.1.

Listing 18.1: Division by zero halts execution with an error message.

```
> 3 / 0;;
System.DivideByZeroException: Attempted to divide by zero.
  at <StartupCode$FSI_0002>.$FSI_0002.main@ () [0x00001] in
   <0e5b9fd12a6649c598d7fa8c09a58dd3>: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,
   System.Reflection.BindingFlags invokeAttr,
   System.Reflection.Binder binder, System.Object[] parameters,
   System.Globalization.CultureInfo culture) [0x00032] in
   <c9f8153c41de4f8cbafd0e32f9bf6b28>:0
Stopped due to error
```

The error message starts by `System.DivideByZeroException: Attempted to divide by zero`, followed by a description of which libraries were involved when the error occurred, and finally F# informs us that it `Stopped due to error`. The type `System.DivideByZeroException` is a built-in exception type, and the built-in integer division operator chooses to raise the exception when the undefined division by zero is attempted. Many times such errors can be avoided by clever program design. However, this is not always possible or desirable, which is why F# implements exception handling for graceful control.

Exceptions are a basic-type called *exn*, and F# has a number of built-in, a few of which are listed in Table 18.1.

· `exn`

Exceptions are handled by the `try`–keyword expressions. We say that an expression may *raise* or *cast* an exception, the `try`–expression may *catch* and *handle* the exception by another expression.

· raising exception
· casting exceptions
· catching exception
· handling exception

Exceptions like in Listing 18.1 may be handled by `try–with` expressions as demonstrated in Listing 18.2.

Attribute	Description
`ArgumentException`	Arguments provided are invalid.
`DivideByZeroException`	Division by zero.
`NotFiniteNumberException`	floating point value is plus or minus infinity, or Not-a-Number (NaN).
`OverflowException`	Arithmetic or casting caused an overflow.
`IndexOutOfRangeException`	Attempting to access an element of an array using an index which is less than zero or equal or greater than the length of the array.

Table 18.1: Some built-in exceptions. The prefix `System.` has been omitted for brevity.

Listing 18.2 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)
```

```
$ fsharpc --nologo exceptionDivByZero.fsx && mono
   exceptionDivByZero.exe
3 / 1 = 3
3 / 0 = 2147483647
```

In the example, when the division operator raises the `System.DivideByZeroException` exception, then `try–with` catches it and returns the value `System.Int32.MaxValue`. Division by zero is still an undefined operation, but with the exception system, the program is able to receive a message about this undefined situation and choose an appropriate action.

The `try` expressions comes in two flavors: `try–with` and `try–finally` expressions.

The *`try–with`* expression has the following syntax,

· `try–with`

Listing 18.3: Syntax for the `try–with` exception handling.

```
try
   <testExpr>
with
   [ | ] <pat1> -> <exprHndl1>
   | <pa2> -> <exprHndl2>
   | <pat3> -> <exprHndl3>
   ...
```

where `<testExpr>` is an expression, which might raise an exception, `<patn>` is a pattern, and `<exprHndln>` is the corresponding exception handler. The value of the `try`–expression is either the value of `<testExpr>`, if it does not raise an exception, or the value of the exception handler `<exprHndln>` of the first matching pattern `<patn>`. The above is lightweight syntax. Regular syntax omits newlines.

In Listing 18.2 *dynamic type matching* is used (see Section 15.9) using the "`:?`" lexeme, i.e., the pattern matches exceptions at runtime which has the `System.DivideByZeroException` type. The exception value may contain furter information and can be accessed if named using the *as*–keyword as demonstrated in Listing 18.4.

· dynamic type pattern

· `as`

Listing 18.4 exceptionDivByZeroNamed.fsx:
Exception value is bound to a name. Compare to Listing 18.2.

```
let div enum denom =
  try
    enum / denom
  with
    | :? System.DivideByZeroException as ex ->
      printfn "Error: %s" ex.Message
      System.Int32.MaxValue

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

```
$ fsharpc --nologo exceptionDivByZeroNamed.fsx
$ mono exceptionDivByZeroNamed.exe
3 / 1 = 3
Error: Attempted to divide by zero.
3 / 0 = 2147483647
```

Here the exception value is bound to the name `ex`.

All exceptions may be caught as the dynamic type `System.Exception`, and F# implements a short-hand for catching exceptions and binding its value to a name as demonstrated in

Listing 18.5 exceptionDivByZeroShortHand.fsx:
An exception of type `System.Exception` is bound to a name. Compare to Listing 18.4.

```
let div enum denom =
  try
    enum / denom
  with
    | ex -> printfn "Error: %s" ex.Message;
   System.Int32.MaxValue

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

```
$ fsharpc --nologo exceptionDivByZeroShortHand.fsx
$ mono exceptionDivByZeroShortHand.exe
3 / 1 = 3
Error: Attempted to divide by zero.
3 / 0 = 2147483647
```

Finally, the short-hand may be guarded with a *when*–guard as demonstrated in Listing 18.6.

· `when`

Listing 18.6 exceptionDivByZeroGuard.fsx:
An exception of type `System.Exception` is bound to a name and guarded. Compare to Listing 18.5.

```
let div enum denom =
  try
    enum / denom
  with
    | ex when enum = 0 -> 0
    | ex -> System.Int32.MaxValue

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

```
$ fsharpc --nologo exceptionDivByZeroGuard.fsx
$ mono exceptionDivByZeroGuard.exe
3 / 1 = 3
3 / 0 = 2147483647
0 / 0 = 0
```

The first pattern only matches the `System.Exception` exception when `enum` is 0, in which case the exception handler returns 0.

Thus, if you don't care about the type of exception, then you need only use the short-hand pattern matching and name binding demonstrated in Listing 18.5 and Listing 18.6, but if you would like to distinguish between types of exceptions, then you must use explicit type matching and possibly value binding demonstrated in Listing 18.2 and Listing 18.4

The *try–finally* expression has the following syntax, · try–finally

Listing 18.7: Syntax for the try–finally exception handling.

```
try
   <testExpr>
finally
   <cleanupExpr>
```

The try–finally expression evaluates the <cleanupExpr> expression following evaluation of the <testExpr> regardless of whether an exception is raised or not as illustrated in Listing 18.8.

Listing 18.8 exceptionDivByZeroFinally.fsx:
The `finally` branch is executed regardless of an exception.

```
let div enum denom =
  printf "Doing division:"
  try
    printf " %d %d." enum denom
    enum / denom
  finally
    printfn " Division finished."

printfn "3 / 1 = %d" (try div 3 1 with ex -> 0)
printfn "3 / 0 = %d" (try div 3 0 with ex -> 0)
```

```
$ fsharpc --nologo exceptionDivByZeroFinally.fsx
$ mono exceptionDivByZeroFinally.exe
Doing division: 3 1. Division finished.
3 / 1 = 3
Doing division: 3 0. Division finished.
3 / 0 = 0
```

Here, the `finally` branch is evaluated following the evaluation of the test expression regardless of whether the test expression raises an exception or not. However, if an exception is raised in a `try–finally` expression and there is no outer `try–with` expression, then execution stops without having evaluated the `finally` branch.

Exceptions can be raised using the *`raise`*-function · `raise`

Listing 18.9: Syntax for the `raise` function that raises exceptions.

```
raise (<expr>)
```

An example of raising the `System.ArgumentException` is shown in Listing 18.10

Listing 18.10 raiseArgumentException.fsx:
Raising the division by zero with customized message.

```
let div enum denom =
  if denom = 0 then
    raise (System.ArgumentException "Error: \"division by 0\"")
  else
    enum / denom

printfn "3 / 0 = %s" (try (div 3 0 |> string) with ex ->
  ex.Message)
```

```
$ fsharpc --nologo raiseArgumentException.fsx
$ mono raiseArgumentException.exe
3 / 0 = Error: "division by 0"
```

In this example, division by zero is never attempted, instead an exception is raised, which must be handled by the caller. Note that the type of `div` is `int -> int -> int` because `denom` 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.

Programs may define new exceptions using the syntax,

Listing 18.11: Syntax for defining new exceptions.

```
exception <ident> of <typeId> {* <typeId>}
```

An example of defining a new exception and raising it is given in Listing 18.12.

Listing 18.12 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 (try picky 3 |> string with ex ->
   ex.Message)
printfn "picky %A = %A" 5 (try picky 5 |> string with ex ->
   ex.Message)
```

```
$ fsharpc --nologo exceptionDefinition.fsx
$ mono exceptionDefinition.exe
picky 3 = "3"
picky 5 = "Exception of type
   'ExceptionDefinition+DontLikeFive' was thrown."
```

Here an exception called `DontLikeFive` is defined, and it is raised in the function `picky`. The example demonstrates that catching the exception as a `System.Exception` as in Listing 18.5 the `Message` property includes information about the exception name but not its argument. To retrieve the argument `"5 sucks"`, we must match the exception with correct exception name as demonstrated in Listing 18.13.

Listing 18.13 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
```

```
$ fsharpc --nologo exceptionDefinitionNCatch.fsx
$ mono exceptionDefinitionNCatch.exe
picky 3 = 3
Exception caught with message: 5 sucks
```

F# includes the *failwith* function to simplify the most common use of exceptions. It is defined as `failwith : string -> exn` and takes a string and raises the built-in `System.Exception` exception. An example of its use is shown in Listing 18.14.

· `failwith`

Listing 18.14 exceptionFailwith.fsx:
An exception raised by `failwith`.

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

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

Unhandled Exception:
System.Exception: hej
  at <StartupCode$exceptionFailwith>.$ExceptionFailwith$fsx.main@ () [0x0000b] in <599574c21515099da7450383c2749559>:0
[ERROR] FATAL UNHANDLED EXCEPTION: System.Exception: hej
  at <StartupCode$exceptionFailwith>.$ExceptionFailwith$fsx.main@ () [0x0000b] in <599574c21515099da7450383c2749559>:0
```

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 shown in Listing 18.15.

Listing 18.15 exceptionSystemException.fsx:
Catching a `failwith` exception using type matching pattern.

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

```
$ fsharpc --nologo exceptionSystemException.fsx

exceptionSystemException.fsx(5,5): warning FS0067: This type
   test or downcast will always hold

exceptionSystemException.fsx(5,5): warning FS0067: This type
   test or downcast will always hold
$ mono exceptionSystemException.exe
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 using the wildcard pattern as in Listing 18.16,

Listing 18.16 exceptionMatchWildcard.fsx:
Catching a `failwith` exception using the wildcard pattern.

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

```
$ fsharpc --nologo exceptionMatchWildcard.fsx
$ mono exceptionMatchWildcard.exe
So failed
```

or use the built-in `Failure` pattern as in Listing 18.17.

Listing 18.17 exceptionFailure.fsx:
Catching a `failwith` exception using the `Failure` pattern.

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

```
$ fsharpc --nologo exceptionFailure.fsx && mono
   exceptionFailure.exe
The castle of "Arrrrg"
```

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

Invalid arguments is such a common reason for failures that a built-in function has been supplied in F#. The ***invalidArg*** takes 2 strings and raises the built-in **ArgumentException** · invalidArg

Listing 18.18 exceptionInvalidArg.fsx:
An exception raised by invalidArg. Compare with Listing 18.10.

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

```
$ fsharpc --nologo exceptionInvalidArg.fsx
$ mono exceptionInvalidArg.exe

Unhandled Exception:
System.ArgumentException: is too much 'a'
Parameter name: a
  at
   <StartupCode$exceptionInvalidArg>.$ExceptionInvalidArg$fsx.main@
   () [0x0000b] in <599574c911642f55a7450383c9749559>:0
[ERROR] FATAL UNHANDLED EXCEPTION: System.ArgumentException:
   is too much 'a'
Parameter name: a
  at
   <StartupCode$exceptionInvalidArg>.$ExceptionInvalidArg$fsx.main@
   () [0x0000b] in <599574c911642f55a7450383c9749559>:0
```

The **invalidArg** function raises an **System.ArgumentException** as shown in Listing 18.19.

Listing 18.19 exceptionInvalidArgNCatch.fsx:
Catching the exception raised by invalidArg.

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

```
$ fsharpc --nologo exceptionInvalidArgNCatch.fsx
$ mono exceptionInvalidArgNCatch.exe
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 re-raise the exception. This can be done with the ***reraise*** as shown in Listing 18.20. · reraise

Listing 18.20 exceptionReraise.fsx:
Reraising an exception.

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

```
$ fsharpc --nologo exceptionReraise.fsx && mono
   exceptionReraise.exe
The castle of "Arrrrg"

Unhandled Exception:
System.Exception: Arrrrg
  at
   <StartupCode$exceptionReraise>.$ExceptionReraise$fsx.main@
   () [0x00041] in <599574d491e0c9eea7450383d4749559>:0
[ERROR] FATAL UNHANDLED EXCEPTION: System.Exception: Arrrrg
  at
   <StartupCode$exceptionReraise>.$ExceptionReraise$fsx.main@
   () [0x00041] in <599574d491e0c9eea7450383d4749559>:0
```

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

18.2 Option types

At exceptions, it is not always obvious what should be returned. E.g., in the Listing 18.2, 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*. 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. An example of rewriting Listing 18.2 to correctly represent the non-computable value is shown in Listing 18.21.

· option type
· `None`
· `Some`

Listing 18.21: 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`*** as illustrated in Listing 18.22.

· `IsNone`
· `IsSome`
· `Value`

Listing 18.22 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
```

```
$ fsharpc --nologo option.fsx && mono option.exe
Some 3 <null>
3 false true
```

The `Value` member is not defined for `None`, thus **explicit pattern matching for extracting values from an option type is preferred, e.g.,** `let get (opt : 'a option) (def : 'a) = match opt with Some x -> x | _ -> def`. Note also that `printf` prints the value `None` as `<null>`. This author hopes, that future versions of the option type will have better visual representations of the `None` value.

Advice

Functions of option types are defined using the ***`option`***–keyword. E.g., to define a function with explicit type annotation that always returns `None`, write `let f (x : 'a option) = None`.

· `option`

F# includes an extensive `Option` module. It defines among many other functions ***`Option.bind`*** which implements `let bind f opt = match opt with None -> None | Some x -> f x`. The function `Option.bind` is demonstrated in Listing 18.23.

`Option.bind`

Listing 18.23: `Option.bind` is useful for cascading calculations on option types.

```
> Option.bind (fun x -> Some (2*x)) (Some 3);;
val it : int option = Some 6
```

The `Option.bind` is a useful tool for cascading functions that evaluates to option types.

18.3 Programming intermezzo: Sequential division of floats

The following problem illustrates cascading error handling:

Problem 18.1

Given a list of floats such as `[1.0; 2.0; 3.0]`, calculate the sequential division `1.0/2.0/3.0`.

A sequential division is safe if the list does not contain zero values. However, if any element in the list is zero, then error handling must be performed. An example using `failwith` is given in Listing 18.24.

Listing 18.24 seqDiv.fsx:
Sequentially dividing a list of numbers.

```
let rec seqDiv acc lst =
  match lst with
    | [] -> acc
    | elm::rest when elm <> 0.0 -> seqDiv (acc/elm) rest
    | _ -> failwith "Division by zero"

try
  printfn "%A" (seqDiv 1.0 [1.0; 2.0; 3.0])
  printfn "%A" (seqDiv 1.0 [1.0; 0.0; 3.0])
with
  Failure msg -> printfn "%s" msg
```

```
$ fsharpc --nologo seqDiv.fsx && mono seqDiv.exe
0.1666666667
Division by zero
```

In this example, a recursive function is defined which updates an accumulator element, initially set to the neutral value 1.0. Division by zero results in a `failwith` exception, wherefore we must wrap its use in a `try—with` expression.

Instead of using exceptions, we may use `Option.bind`. In order to use `Option.bind` for a sequence of non-option floats, we will define a division operator, that reverses the order of operands. This is shown in Listing 18.25.

Listing 18.25 seqDivOption.fsx:
Sequentially dividing a sequence of numbers using `Option.bind`. Compare with Listing 18.24.

```
let divideBy denom enum =
  if denom = 0.0 then
    None
  else
    Some (enum/denom)

let success =
  Some 1.0
  |> Option.bind (divideBy 2.0)
  |> Option.bind (divideBy 3.0)
printfn "%A" success

let fail =
  Some 1.0
  |> Option.bind (divideBy 0.0)
  |> Option.bind (divideBy 3.0)
printfn "%A; isNone: %b" fail fail.IsNone
```

```
$ fsharpc --nologo seqDivOption.fsx && mono seqDivOption.exe
Some 0.1666666667
<null>; isNone: true
```

Here the function `divideBy` takes two non-option arguments and returns an option type. Thus, `Option.bind (divideBy 2.0) (Some 1.0)` is equal to `Some 0.5`, since `divideBy 2.0` is a function that divides any float argument by 2.0. Iterating `Option.bind (divideBy 3.0) (Some 0.5)` we calculate `Some 0.1666666667` or `Some (1.0/6.0)` as expected. In Listing 18.25 this is written as a single `let` binding using piping. And since `Option.bind` correctly handles the distinction between `Some` and `None` values, such piping sequences correctly handles possible errors as shown in Listing 18.25.

The sequential application can be extended to lists using `List.foldBack` as demonstrated in Listing 18.26.

Listing 18.26 seqDivOptionAdv.fsx:
Sequentially dividing a list of numbers using `Option.bind` and `List.foldBack`. Compare with Listing 18.25.

```
let divideBy denom enum =
  if denom = 0.0 then
    None
  else
    Some (enum/denom)
let divideByOption x acc =
  Option.bind (divideBy x) acc

let success = List.foldBack divideByOption [3.0; 2.0; 1.0] (Some 1.0)
printfn "%A" success

let fail = List.foldBack divideByOption [3.0; 0.0; 1.0] (Some 1.0)
printfn "%A; isNone: %A" fail fail.IsNone
```

```
$ fsharpc --nologo seqDivOptionAdv.fsx && mono seqDivOptionAdv.exe
Some 0.1666666667
<null>; isNone: true
```

Since `List.foldBack` processes the list from the right, the list of integers have been reversed. Notice how `divideByOption` is the function spelled out in each piping step of Listing 18.25.

Exceptions and option type are systems to communicate errors up through a hierarchy of function calls. While exceptions favor imperative style programming, option types are functional style programming. Exceptions allow for a detailed report of the type of error to the caller, whereas option types only allow for flagging that an error has occurred.

19 | Working with files

An important part of programming is handling data. A typical source of data is 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 23, 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. For example, the `printf` family of functions discussed in Section 6.5 is defined in the `Printf` module of the `Fsharp.Core` namespace, and it is used to put characters on the `stdout` stream, i.e., to print on the screen. Likewise, `ReadLine` discussed in Section 6.6 is defined in the `System.Console` class, and it fetches characters from the `stdin` stream, that is, reads the characters the user types on the keyboard until the newline is pressed.

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 hard disk, 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 between 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 in a file in a human-readable form and interpreted from UTF8 when retrieving them at a later point from the 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

· create file
· open file
· read file
· write file
· close file
· delete file
· stream

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.[1]

[1] Jon: **Maybe add a figure illustrating the difference between files and streams.**

19.1 Command line arguments

Compiled programs may be started from the console with one or more arguments. E.g., if we have made a program called `prog`, then arguments may be passed as `mono prog arg1 arg2 ...`. To read the arguments in the program must define a function with the *EntryPoint* attribute, and this function must be of type `string array -> int`.

· EntryPoint

Listing 19.1: Defining an entry point function with arguments from the console.

```
[<EntryPoint>]
let <funcIdent> <arg> =
  <bodyExpr>
```

`<funcIdent>` is the function's name, `<arg>` is the name of an array of strings, and `<bodyExpr>` is the function body. Return value 0 implies a successful execution of the program, while non-zero means failure. The entry point function must be in the last file compiled. An example is given in Listing 19.2.

Listing 19.2 commandLineArgs.fsx:
Interacting with a user with `ReadLine` and `WriteLine`.

```
[<EntryPoint>]
let main args =
    printfn "Arguments passed to function : %A" args
    0  // Signals that program terminated successfully
```

An example execution with arguments is shown in Listing 19.3.

Listing 19.3: An example dialogue of running Listing 19.2.

```
$ fsharpc --nologo commandLineArgs.fsx
$ mono commandLineArgs.exe Hello World
Arguments passed to function : [|"Hello"; "World"|]
```

In Bash, the return values is called the *exit status* and can be tested using Bash's `if` statements as demonstrated in Listing 19.4.

· exit status

Listing 19.4: Testing return values in Bash when running Listing 19.2.

```
$ fsharpc --nologo commandLineArgs.fsx
$ if mono commandLineArgs.exe Hello World; then echo
  "success"; else echo "failure"; fi
Arguments passed to function : [|"Hello"; "World"|]
success
```

Also in Bash, the exit status of the last executed program can be accessed using the `$?` built-in environment variable. In Windows this same variable is called `%errorlevel%`.

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.

Table 19.1: Three built-in streams in `System.Console`.

Function	Description
`Write: string -> unit`	Write to the console. E.g., `System.Console.Write "Hello world"`. Similar to `printf`.
`WriteLine: string -> unit`	As `Write` but followed by a newline character, e.g., `WriteLine "Hello world"`. Similar to `printfn`.
`Read: unit -> int`	Read the next key from the keyboard blocking execution as long, e.g., `Read ()`. The key pressed is echoed to the screen.
`ReadKey: bool -> System.ConsoleKeyInfo`	As `Read` but returns more information about the key pressed. When given the value `true` as argument, then the key pressed is not echoed to the screen. E.g., `ReadKey true`.
`ReadLine unit -> string`	Read the next sequence of characters until newline from the keyboard, e.g., `ReadLine ()`.

Table 19.2: Some functions for interacting with the user through the console in the `System.Console` class. Prefix "`System.Console.`" is omitted for brevity.

19.2 Interacting with the console

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 three built-in streams in `System.Console` listed in Table 19.1. 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 shown in Table 19.2. 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. A demonstration of the use of `Write`, `WriteLine`, and `ReadLine` is given in Listing 19.5.

Listing 19.5 userDialogue.fsx:
Interacting with a user with `ReadLine` and `WriteLine`. The user typed "3.5" and "7.4".

```
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))
```

```
$ fsharpc --nologo userDialogue.fsx && mono userDialogue.exe
To perform the multiplication of a and b
Enter a: 3.5
Enter b: 7.4
a * b = 25.9
```

The functions `Write` and `WriteLine` acts as `printfn` but without a formatting string. These functions have many overloaded definitions the description of which is outside the scope of this book. **For writing to the console, `printf` is to be preferred.** Advice

Often `ReadKey` is preferred over `Read`, since the former returns a value of type `System.ConsoleKeyInfo`, which is a structure with three properties:

`Key` A `System.ConsoleKey` enumeration of the key pressed. E.g., the character 'a' is `ConsoleKey.A`.

`KeyChar` A unicode representation of the key.

`Modifiers` A `System.ConsoleModifiers` enumeration of modifier keys shift, crtl, and alt.

An example of a dialogue is shown in Listing 19.6.

Listing 19.6 readKey.fsx:
Reading keys and modifiers. The user press 'a', 'shift-a', and 'crtl-a', and the program was terminated by pressing 'crtl-c'. The 'alt-a' combination does not work on MacOS.

```
open System

printfn "Start typing"
while true do
  let key = Console.ReadKey true
  let shift =
    if key.Modifiers = ConsoleModifiers.Shift then "SHIFT+"
   else ""
  let alt =
    if key.Modifiers = ConsoleModifiers.Alt then "ALT+" else ""
  let ctrl =
    if key.Modifiers = ConsoleModifiers.Control then "CTRL+"
   else ""
  printfn "You pressed: %s%s%s%s" shift alt ctrl
   (key.Key.ToString ())
```

```
$ fsharpc --nologo readKey.fsx && mono readKey.exe
Start typing
You pressed: A
You pressed: SHIFT+A
You pressed: CTRL+A
```

19.3 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. While a file is open, the operating system will protect it depending on how the file is opened. 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. 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.** 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

Advice

Data in 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

System.IO.File	Description
`Open:` `(path : string) * (mode : FileMod` `-> FileStream`	Request the opening of a file on `path` for reading and writing with access mode `FileMode`, see Table 19.4. 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 19.3: The family of `System.IO.File.Open` functions. See Table 19.4 for a description of `FileMode`, Tables 19.5 and 19.6 for a description of `FileStream`, Table 19.7 for a description of `StreamReader`, and Table 19.8 for a description of `StreamWriter`.

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 19.3.

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 19.4. An example of how a file is opened and later closed is shown in Listing 19.7.

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 19.4: File mode values for the `System.IO.Open` function.

Listing 19.7 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 ()
```

```
$ fsharpc --nologo openFile.fsx && mono openFile.exe
The file "openFile.fsx" was successfully opened.
```

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 F# the distinction between files and streams are not very clear. F# 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 19.3, 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 19.5.
This information is important in order to restrict the operation that we will perform on the file. Some typical operations are listed in and 19.6. 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

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 19.5: 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 19.6: Some methods of the `System.IO.FileStream` class.

is useful to be able to force the operating system to empty its buffer to the device. This is called *flushing* and can be forced using the `Flush` function. · flushing

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 19.7. 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 19.8.

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 19.7: 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 19.8: Some methods of the `System.IO.StreamWriter` class.

Listing 19.8 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
```

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

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 19.9: 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`.
`SetCurrentDirectory : (path : string) -> unit`	Set the current working directory of the program to `path`.

Table 19.10: Some methods of the `System.IO.Directory` class.

19.4 Working with files and directories.

F# has support for managing files summarized in the `System.IO.File` class and summarized in Table 19.9.

In the `System.IO.Directory` class there are a number of other frequently used functions summarized in Table 19.10.

In the `System.IO.Path` class there are a number of other frequently used functions summarized in Table 19.11.

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

· 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 19.11: Some methods of the `System.IO.Path` class.

filename.

F#'s `System` namespace contains functions for accessing web pages as stream as illustrated in Listing 19.9.

Listing 19.9 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]
```

```
$ fsharpc --nologo webRequest.fsx && mono webRequest.exe
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 *Uni-*

form 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 19.7 to access the web page. [2]

· Uniform Resource Identifiers
· URI

19.6 Resource Management

Streams and files are examples of computer resources, that may be shared by several applications. Most operating systems allow for several applications to be running in parallel, and to avoid unnecessarily blocking and hogging of resources, all responsible applications must release resources as soon as they are done using them. F# has language constructions for automatic releasing of resources: the use binding and the using function. These automatically dispose of resources, when the resource's name binding falls out of scope. Technically this is done by calling the *Dispose* method on objects that implement the *System.IDisposable* interface. See Section 21.3 for more on interfaces.

· Dispose
· System.IDisposable

The *use* keyword is similar to let:

· use

Listing 19.10: Use binding expression.

```
use <valueIdent> = <bodyExpr> [in <expr>]
```

A use binding provides a binding between the <bodyExpr> expression to the name <valueIdent> in the following expression(s), and adds a call to Dispose() on <valueIdent> if it implements System.IDisposable. See for example Listing 19.11.

Listing 19.11 useBinding.fsx:
Using use instead of let releases disposable resources at end of scope.

```
open System.IO

let writeToFile (filename : string) (str : string) : unit =
  use file = File.CreateText filename
  file.Write str
  // file.Dispose() is implicitely called here,
  // implying that the file is closed.

writeToFile "use.txt" "Using 'use' closes the file, when out
  of scope."
```

Here, file is an System.IDisposable object, and before writeToFile returns, file.Dispose() is called automatically, which in this case implies that the file is closed. Had we used let instead, then the file would first be closed, when the program terminates.

The higher order function *using* takes a disposable object and a function, executes the function on the disposable objects and then calls Dispose() on the disposable object. This is illustrated in Listing 19.12

· using

[2]Jon: **This section could be extended...**

Listing 19.12 using.fsx:
The using function executes a function on an object and releases its disposable resources. Compare with Listing 19.11.

```
open System.IO

let writeToFile (str : string) (file : StreamWriter) : unit =
    file.Write str

using (File.CreateText "use.txt") (writeToFile "Disposed after call.")
// Dispose() is implicitely called on the anonymous file handle implying
// that the file is automatically closed.
```

The main difference between `use` and `using` is that resources allocated using `use` are disposed at the end of its scope, while `using` disposes the resources after the execution of the function in its argument. In spite of the added control of `using`, we **prefer `use` over `using` due to its simpler structure.**

Advice

19.7 Programming intermezzo: Ask user for existing file

A typical problem, when working with files, is

Problem 19.1
Ask the user for the name of an existing file.

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 19.13 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
```

Bibliography

[1] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345—-363, 1936.

[2] Ole-Johan Dahl and Kristen Nygaard. SIMULA a language for programming and description of discrete event systems. introduction and user's manual. Technical report, Norwegian Computing Center, 1967.

[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.

[10] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1936.

Index