1 Handling Errors and Exceptions

1.1 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 1.1.

Listing 1.1: 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# states that it `Stopped due to error`. `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 ones, a few of which are listed in Table 1.1.

· exn@`exn`

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

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

Exceptions like in Listing 1.1 may be handled by `try`–`with` expressions, as demonstrated in Listing 1.2.

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

Listing 1.2 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, · trywith@`try–with`

Listing 1.3: 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 using lightweight syntax. Regular syntax omits newlines.

In Listing 1.2 *dynamic type matching* is used (see **??**) using the "`:?`" lexeme, i.e., the pattern matches exception with type `System.DivideByZeroException` at runtime. The exception value may contain furter information and can be accessed if named using the *as*–keyword, as demonstrated in Listing 1.4.

· dynamic type pattern

· as@as

Listing 1.4 exceptionDivByZeroNamed.fsx:
Exception value is bound to a name. Compare to Listing 1.2.

```
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 an exception and binding its value to a name as demonstrated in Listing 1.5

Listing 1.5 exceptionDivByZeroShortHand.fsx:
An exception of type `System.Exception` is bound to a name. Compare to Listing 1.4.

```
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 1.6. · when@when

Listing 1.6 exceptionDivByZeroGuard.fsx:
An exception of type `System.Exception` is bound to a name and guarded. Compare to Listing 1.5.

```
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 1.5 and Listing 1.6, 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 1.2 and Listing 1.4

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

Listing 1.7: 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 1.8.

Listing 1.8 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@`raise`

Listing 1.9: Syntax for the `raise` function that raises exceptions.

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

An example of raising the `System.ArgumentException` is shown in Listing 1.10

Listing 1.10 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 and 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 raise exceptions are ignored, and the type is inferred by the remaining branches.

Programs may define new exceptions using the syntax,

Listing 1.11: Syntax for defining new exceptions.

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

An example of defining a new exception and raising it is given in Listing 1.12.

Listing 1.12 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 1.5, 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 the correct exception name, as demonstrated in Listing 1.13.

**Listing 1.13 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 1.14.

· failwith@`failwith`

**Listing 1.14 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 several choices. The exception casts a `System.Exception` exception, which may be caught using the `:?` pattern, as shown in Listing 1.15.

Listing 1.15 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 match the type of `System.Exception`. Instead, it is better to either match using the wildcard pattern as in Listing 1.16,

Listing 1.16 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 1.17.

Listing 1.17 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 an argument.

Invalid arguments are such a common reason for failures, that a built-in function for handling them has been supplied in F#. The ***`invalidArg`*** takes 2 strings and raises the built-in `ArgumentException`, as shown in Listing 1.18. · invalidArg@**`invalidA`**

Listing 1.18 exceptionInvalidArg.fsx:
An exception raised by `invalidArg`. Compare with Listing 1.10.

```
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 1.19.

Listing 1.19 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 1.20. · reraise@**`reraise`**

**Listing 1.20 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.

1.2 Option Types

At exceptions, it is not always obvious what should be returned. E.g., in the Listing 1.2, the exception is handled gracefully, but the return value is somewhat arbitrarily chosen to be the largest possible integer. Instead, we may 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 1.2 to correctly represent the non-computable value is shown in Listing 1.21.

· option type

· none@`None`

· some@`Some`

Listing 1.21: 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 and tested for by its member functions, ***IsNone***, ***IsSome***, and ***Value***, as illustrated in Listing 1.22.

· isnone@IsNone
· issome@IsSome
· value@Value

Listing 1.22 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 it is adviced to **prefer explicit pattern matching for extracting values from an option type.** An example is: `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 on 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@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 1.23.

· option.bind@Option.

Listing 1.23: Using `Option.bind` to perform 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.

1.3 Programming Intermezzo: Sequential Division of Floats

The following problem illustrates cascading error handling:

Problem 1.1

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 1.24.

Listing 1.24 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 1.25.

Listing 1.25 seqDivOption.fsx:
Sequentially dividing a sequence of numbers using `Option.bind`. Compare with Listing 1.24.

```
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 1.25, 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 handle possible errors, as shown in Listing 1.25.

The sequential application can be extended to lists, using `List.foldBack`, as demonstrated in Listing 1.26.

Listing 1.26 seqDivOptionAdv.fsx:
Sequentially dividing a list of numbers, using `Option.bind` and `List.foldBack`. Compare with Listing 1.25.

```
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 has been reversed. Notice how `divideByOption` is the function spelled out in each piping step of Listing 1.25.

Exceptions and option type are systems to communicate errors up through a hierarchy of function calls. While exceptions favor imperative style programming, option types belong to 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.