8 Error Handling

Nobody likes dealing with errors. It's tedious, it's easy to get wrong, and it's usually just not as fun as thinking about how your program is going to succeed. But error handling is important, and however much you don't like thinking about it, having your software fail due to poor error handling is worse.

Thankfully, OCaml has powerful tools for handling errors reliably and with a minimum of pain. In this chapter we'll discuss some of the different approaches in OCaml to handling errors, and give some advice on how to design interfaces that make error handling easier.

We'll start by describing the two basic approaches for reporting errors in OCaml: error-aware return types and exceptions.

8.1 Error-Aware Return Types

The best way in OCaml to signal an error is to include that error in your return value. Consider the type of the `find` function in the `List` module:

```
# open Base;;
# List.find;;
- : 'a list -> f:('a -> bool) -> 'a option = <fun>
```

The option in the return type indicates that the function may not succeed in finding a suitable element:

```
# List.find [1;2;3] ~f:(fun x -> x >= 2);;
- : int option = Some 2
# List.find [1;2;3] ~f:(fun x -> x >= 10);;
- : int option = None
```

Including errors in the return values of your functions requires the caller to handle the error explicitly, allowing the caller to make the choice of whether to recover from the error or propagate it onward.

Consider the `compute_bounds` function below, which takes a list and a comparison function and returns upper and lower bounds for the list by finding the smallest and largest element on the list. `List.hd` and `List.last`, which return `None` when they encounter an empty list, are used to extract the largest and smallest element of the list:

```
# let compute_bounds ~compare list =
    let sorted = List.sort ~compare list in
    match List.hd sorted, List.last sorted with
    | None,_ | _, None -> None
    | Some x, Some y -> Some (x,y);;
val compute_bounds : compare:('a -> 'a -> int) -> 'a list -> ('a * 'a)
    option =
  <fun>
```

The match expression is used to handle the error cases, propagating a None in hd or last into the return value of compute_bounds.

On the other hand, in the find_mismatches that follows, errors encountered during the computation do not propagate to the return value of the function. find_mismatches takes two hash tables as arguments and searches for keys that have different data in one table than in the other. As such, the failure to find a key in one table isn't a failure of any sort:

```
# let find_mismatches table1 table2 =
    Hashtbl.fold table1 ~init:[] ~f:(fun ~key ~data mismatches ->
      match Hashtbl.find table2 key with
      | Some data' when data' <> data -> key :: mismatches
      | _ -> mismatches
    );;
val find_mismatches :
  ('a, int) Hashtbl.Poly.t -> ('a, int) Hashtbl.Poly.t -> 'a list =
    <fun>
```

The use of options to encode errors underlines the fact that it's not clear whether a particular outcome, like not finding something on a list, is an error or is just another valid outcome. This depends on the larger context of your program, and thus is not something that a general-purpose library can know in advance. One of the advantages of error-aware return types is that they work well in both situations.

8.1.1 Encoding Errors with Result

Options aren't always a sufficiently expressive way to report errors. Specifically, when you encode an error as None, there's nowhere to say anything about the nature of the error.

Result.t is meant to address this deficiency. The type is defined as follows:

```
module Result : sig
   type ('a,'b) t = | Ok of 'a
                    | Error of 'b
end
```

A Result.t is essentially an option augmented with the ability to store other information in the error case. Like Some and None for options, the constructors Ok and Error are available at the toplevel. As such, we can write:

```
# [ Ok 3; Error "abject failure"; Ok 4 ];;
- : (int, string) result list = [Ok 3; Error "abject failure"; Ok 4]
```

without first opening the Result module.

8.1.2 Error and Or_error

Result.t gives you complete freedom to choose the type of value you use to represent errors, but it's often useful to standardize on an error type. Among other things, this makes it easier to write utility functions to automate common error handling patterns.

But which type to choose? Is it better to represent errors as strings? Some more structured representation like XML? Or something else entirely?

Base's answer to this question is the Error.t type. You can, for example, construct one from a string.

```
# Error.of_string "something went wrong";;
- : Error.t = something went wrong
```

An Or_error.t is simply a Result.t with the error case specialized to the Error.t type. Here's an example.

```
# Error (Error.of_string "failed!");;
- : ('a, Error.t) result = Error failed!
```

The Or_error module provides a bunch of useful operators for constructing errors. For example, Or_error.try_with can be used for catching exceptions from a computation.

```
# let float_of_string s =
    Or_error.try_with (fun () -> Float.of_string s);;
val float_of_string : string -> float Or_error.t = <fun>
# float_of_string "3.34";;
- : float Or_error.t = Base__.Result.Ok 3.34
# float_of_string "a.bc";;
- : float Or_error.t =
Base__.Result.Error (Invalid_argument "Float.of_string a.bc")
```

Perhaps the most common way to create Error.ts is using *s-expressions*. An s-expression is a balanced parenthetical expression where the leaves of the expressions are strings. Here's a simple example:

```
(This (is an) (s expression))
```

S-expressions are supported by the Sexplib package that is distributed with Base and is the most common serialization format used in Base. Indeed, most types in Base come with built-in s-expression converters.

```
# Error.create "Unexpected character" 'c' Char.sexp_of_t;;
- : Error.t = ("Unexpected character" c)
```

We're not restricted to doing this kind of error reporting with built-in types. As we'll discuss in more detail in Chapter 21 (Data Serialization with S-Expressions), Sexplib comes with a syntax extension that can autogenerate sexp converters for specific types. We can enable it in the toplevel with a #require statement enabling ppx_jane, which is a package that pulls in multiple different syntax extensions, including ppx_sexp_value, the one we need here. (Because of technical issues with the toplevel, we can't easily enable these syntax extensions individually.)

```
# #require "ppx_jane";;
# Error.t_of_sexp
    [%sexp ("List is too long",[1;2;3] : string * int list)];;
- : Error.t = ("List is too long" (1 2 3))
```

Error also supports operations for transforming errors. For example, it's often useful to augment an error with information about the context of the error or to combine multiple errors together. Error.tag and Error.of_list fulfill these roles:

```
# Error.tag
    (Error.of_list [ Error.of_string "Your tires were slashed";
                     Error.of_string "Your windshield was smashed" ])
    ~tag:"over the weekend";;
- : Error.t =
("over the weekend" "Your tires were slashed" "Your windshield was
    smashed")
```

A very common way of generating errors is the %message syntax extension, which provides a compact syntax for providing a string describing the error, along with further values represented as s-expressions. Here's an example.

```
# let a = "foo" and b = ("foo",[3;4]);;
val a : string = "foo"
val b : string * int list = ("foo", [3; 4])
# Or_error.error_s
    [%message "Something went wrong" (a:string) (b: string * int
    list)];;
- : 'a Or_error.t =
Base__.Result.Error ("Something went wrong" (a foo) (b (foo (3 4))))
```

This is the most common idiom for generating Error.t's.

8.1.3 bind and Other Error Handling Idioms

As you write more error handling code in OCaml, you'll discover that certain patterns start to emerge. A number of these common patterns have been codified by functions in modules like Option and Result. One particularly useful pattern is built around the function bind, which is both an ordinary function and an infix operator >>=. Here's the definition of bind for options:

```
# let bind option ~f =
    match option with
    | None -> None
    | Some x -> f x;;
val bind : 'a option -> f:('a -> 'b option) -> 'b option = <fun>
```

As you can see, bind None f returns None without calling f, and bind (Some x) ~f returns f x. bind can be used as a way of sequencing together error-producing functions so that the first one to produce an error terminates the computation. Here's a rewrite of compute_bounds to use a nested series of binds:

```
# let compute_bounds ~compare list =
    let sorted = List.sort ~compare list in
    Option.bind (List.hd sorted) ~f:(fun first ->
      Option.bind (List.last sorted) ~f:(fun last ->
         Some (first,last)));;
val compute_bounds : compare:('a -> 'a -> int) -> 'a list -> ('a * 'a)
    option =
  <fun>
```

The preceding code is a little bit hard to swallow, however, on a syntactic level. We can make it easier to read and drop some of the parentheses, by using the infix operator form of `bind`, which we get access to by locally opening `Option.Monad_infix`. The module is called `Monad_infix` because the `bind` operator is part of a subinterface called `Monad`, which we'll see again in Chapter 17 (Concurrent Programming with Async).

```
# let compute_bounds ~compare list =
    let open Option.Monad_infix in
    let sorted = List.sort ~compare list in
    List.hd sorted   >>= fun first ->
    List.last sorted >>= fun last  ->
    Some (first,last);;
val compute_bounds : compare:('a -> 'a -> int) -> 'a list -> ('a * 'a)
    option =
  <fun>
```

This use of `bind` isn't really materially better than the one we started with, and indeed, for small examples like this, direct matching of options is generally better than using `bind`. But for large, complex examples with many stages of error handling, the `bind` idiom becomes clearer and easier to manage.

Monads and `Let_syntax`

We can make this look a little bit more ordinary by using a syntax extension that's designed specifically for monadic binds, called `Let_syntax`. Here's what the above example looks like using this extension.

```
# #require "ppx_let";;
# let compute_bounds ~compare list =
    let open Option.Let_syntax in
    let sorted = List.sort ~compare list in
    let%bind first = List.hd sorted in
    let%bind last  = List.last sorted in
    Some (first,last);;
val compute_bounds : compare:('a -> 'a -> int) -> 'a list -> ('a * 'a)
    option =
  <fun>
```

Note that we needed a `#require` statement to enable the extension.

To understand what's going on here, you need to know that `let%bind x = some_expr in some_other_expr` is rewritten into `some_expr >>= fun x -> some_other_expr`.

The advantage of `Let_syntax` is that it makes monadic bind look more like a regular let-binding. This works nicely because you can think of the monadic bind in this case as a special form of let binding that has some built-in error handling semantics.

There are other useful idioms encoded in the functions in `Option`. One example is `Option.both`, which takes two optional values and produces a new optional pair that is None if either of its arguments are None. Using Option.both, we can make compute_bounds even shorter:

```
# let compute_bounds ~compare list =
    let sorted = List.sort ~compare list in
    Option.both (List.hd sorted) (List.last sorted);;
val compute_bounds : compare:('a -> 'a -> int) -> 'a list -> ('a * 'a)
    option =
  <fun>
```

These error-handling functions are valuable because they let you express your error handling both explicitly and concisely. We've only discussed these functions in the context of the Option module, but more functionality of this kind can be found in the Result and Or_error modules.

8.2 Exceptions

Exceptions in OCaml are not that different from exceptions in many other languages, like Java, C#, and Python. Exceptions are a way to terminate a computation and report an error, while providing a mechanism to catch and handle (and possibly recover from) exceptions that are triggered by subcomputations.

You can trigger an exception by, for example, dividing an integer by zero:

```
# 3 / 0;;
Exception: Division_by_zero.
```

And an exception can terminate a computation even if it happens nested somewhere deep within it:

```
# List.map ~f:(fun x -> 100 / x) [1;3;0;4];;
Exception: Division_by_zero.
```

If we put a printf in the middle of the computation, we can see that List.map is interrupted partway through its execution, never getting to the end of the list:

```
# List.map ~f:(fun x -> Stdio.printf "%d\n%!" x; 100 / x) [1;3;0;4];;
1
3
0
Exception: Division_by_zero.
```

In addition to built-in exceptions like Divide_by_zero, OCaml lets you define your own:

```
# exception Key_not_found of string;;
exception Key_not_found of string
# raise (Key_not_found "a");;
Exception: Key_not_found("a").
```

Exceptions are ordinary values and can be manipulated just like other OCaml values:

```
# let exceptions = [ Division_by_zero; Key_not_found "b" ];;
val exceptions : exn list = [Division_by_zero; Key_not_found("b")]
# List.filter exceptions  ~f:(function
    | Key_not_found _ -> true
    | _ -> false);;
- : exn list = [Key_not_found("b")]
```

Exceptions are all of the same type, exn, which is itself something of a special case in the OCaml type system. It is similar to the variant types we encountered in Chapter 7 (Variants), except that it is *open*, meaning that it's not fully defined in any one place. In particular, new tags (specifically, new exceptions) can be added to it by different parts of the program. This is in contrast to ordinary variants, which are defined with a closed universe of available tags. One result of this is that you can never have an exhaustive match on an exn, since the full set of possible exceptions is not known.

The following function uses the Key_not_found exception we defined above to signal an error:

```
# let rec find_exn alist key = match alist with
    | [] -> raise (Key_not_found key)
    | (key',data) :: tl -> if String.(=) key key' then data else
    find_exn tl key;;
val find_exn : (string * 'a) list -> string -> 'a = <fun>
# let alist = [("a",1); ("b",2)];;
val alist : (string * int) list = [("a", 1); ("b", 2)]
# find_exn alist "a";;
- : int = 1
# find_exn alist "c";;
Exception: Key_not_found("c").
```

Note that we named the function find_exn to warn the user that the function routinely throws exceptions, a convention that is used heavily in Base.

In the preceding example, raise throws the exception, thus terminating the computation. The type of raise is a bit surprising when you first see it: [raise]{.idx}

```
# raise;;
- : exn -> 'a = <fun>
```

The return type of 'a makes it look like raise manufactures a value to return that is completely unconstrained in its type. That seems impossible, and it is. Really, raise has a return type of 'a because it never returns at all. This behavior isn't restricted to functions like raise that terminate by throwing exceptions. Here's another example of a function that doesn't return a value:

```
# let rec forever () = forever ();;
val forever : unit -> 'a = <fun>
```

forever doesn't return a value for a different reason: it's an infinite loop.

This all matters because it means that the return type of raise can be whatever it needs to be to fit into the context it is called in. Thus, the type system will let us throw an exception anywhere in a program.

Declaring Exceptions Using [@@deriving sexp]

OCaml can't always generate a useful textual representation of an exception. For example:

```
# type 'a bounds = { lower: 'a; upper: 'a };;
```

```
type 'a bounds = { lower : 'a; upper : 'a; }
# exception Crossed_bounds of int bounds;;
exception Crossed_bounds of int bounds
# Crossed_bounds { lower=10; upper=0 };;
- : exn = Crossed_bounds(_)
```

But if we declare the exception (and the types it depends on) using `[@@deriving sexp]`, we'll get something with more information:

```
# type 'a bounds = { lower: 'a; upper: 'a } [@@deriving sexp];;
type 'a bounds = { lower : 'a; upper : 'a; }
val bounds_of_sexp : (Sexp.t -> 'a) -> Sexp.t -> 'a bounds = <fun>
val sexp_of_bounds : ('a -> Sexp.t) -> 'a bounds -> Sexp.t = <fun>
# exception Crossed_bounds of int bounds [@@deriving sexp];;
exception Crossed_bounds of int bounds
# Crossed_bounds { lower=10; upper=0 };;
- : exn = (//toplevel//.Crossed_bounds ((lower 10) (upper 0)))
```

The period in front of `Crossed_bounds` is there because the representation generated by `[@@deriving sexp]` includes the full module path of the module where the exception in question is defined. In this case, the string `//toplevel//` is used to indicate that this was declared at the utop prompt, rather than in a module.

This is all part of the support for s-expressions provided by the Sexplib library and syntax extension, which is described in more detail in Chapter 21 (Data Serialization with S-Expressions).

8.2.1 Helper Functions for Throwing Exceptions

Base provides a number of helper functions to simplify the task of throwing exceptions. The simplest one is `failwith`, which could be defined as follows:

```
# let failwith msg = raise (Failure msg);;
val failwith : string -> 'a = <fun>
```

There are several other useful functions for raising exceptions, which can be found in the API documentation for the `Common` and `Exn` modules in Base.

Another important way of throwing an exception is the `assert` directive. `assert` is used for situations where a violation of the condition in question indicates a bug. Consider the following piece of code for zipping together two lists:

```
# let merge_lists xs ys ~f =
    if List.length xs <> List.length ys then None
    else
      let rec loop xs ys =
        match xs,ys with
        | [],[] -> []
        | x::xs, y::ys -> f x y :: loop xs ys
        | _ -> assert false
      in
      Some (loop xs ys);;
val merge_lists : 'a list -> 'b list -> f:('a -> 'b -> 'c) -> 'c list
    option =
  <fun>
# merge_lists [1;2;3] [-1;1;2] ~f:(+);;
- : int list option = Some [0; 3; 5]
# merge_lists [1;2;3] [-1;1] ~f:(+);;
- : int list option = None
```

Here we use `assert false`, which means that the `assert`, once reached, is guaranteed to trigger. In general, one can put an arbitrary condition in the assertion.

In this case, the `assert` can never be triggered because we have a check that makes sure that the lists are of the same length before we call `loop`. If we change the code so that we drop this test, then we can trigger the `assert`:

```
# let merge_lists xs ys ~f =
    let rec loop xs ys =
      match xs,ys with
      | [],[] -> []
      | x::xs, y::ys -> f x y :: loop xs ys
      | _ -> assert false
    in
    loop xs ys;;
val merge_lists : 'a list -> 'b list -> f:('a -> 'b -> 'c) -> 'c list =
  <fun>
# merge_lists [1;2;3] [-1] ~f:(+);;
Exception: "Assert_failure //toplevel//:6:14"
```

This shows what's special about `assert`: it captures the line number and character offset of the source location from which the assertion was made.

8.2.2 Exception Handlers

So far, we've only seen exceptions fully terminate the execution of a computation. But sometimes, we want a program to be able to respond to and recover from an exception. This is achieved through the use of *exception handlers*.

In OCaml, an exception handler is declared using a `try/with` expression. Here's the basic syntax.

```
try <expr> with
| <pat1> -> <expr1>
| <pat2> -> <expr2>
...
```

A `try/with` clause first evaluates its body, *expr*. If no exception is thrown, then the result of evaluating the body is what the entire `try/with` clause evaluates to.

But if the evaluation of the body throws an exception, then the exception will be fed to the pattern-match clauses following the `with`. If the exception matches a pattern, then we consider the exception caught, and the `try/with` clause evaluates to the expression on the right-hand side of the matching pattern.

Otherwise, the original exception continues up the stack of function calls, to be handled by the next outer exception handler. If the exception is never caught, it terminates the program.

8.2.3 Cleaning Up in the Presence of Exceptions

One headache with exceptions is that they can terminate your execution at unexpected places, leaving your program in an awkward state. Consider the following function for loading a file full of numerical data. This code parses data that matches a simple comma-separated file format, where each field is a floating point number. In this example we open `Stdio`, to get access to routines for reading from files.

```
# open Stdio;;
# let parse_line line =
    String.split_on_chars ~on:[','] line
    |> List.map ~f:Float.of_string;;
val parse_line : string -> float list = <fun>
# let load filename =
    let inc = In_channel.create filename in
    let data =
      In_channel.input_lines inc
      |> List.map ~f:parse_line
    in
    In_channel.close inc;
    data;;
val load : string -> float list list = <fun>
```

One problem with this code is that the parsing function can throw an exception if the file in question is malformed. Unfortunately, that means that the `In_channel.t` that was opened will never be closed, leading to a file-descriptor leak.

We can fix this using Base's `Exn.protect` function, which takes two arguments: a thunk `f`, which is the main body of the computation to be run; and a thunk `finally`, which is to be called when `f` exits, whether it exits normally or with an exception. This is similar to the `try/finally` construct available in many programming languages, but it is implemented in a library, rather than being a built-in primitive. Here's how it could be used to fix our `load` function:

```
# let load filename =
    let inc = In_channel.create filename in
    Exn.protect
      ~f:(fun () -> In_channel.input_lines inc |> List.map
    ~f:parse_line)
      ~finally:(fun () -> In_channel.close inc);;
val load : string -> float list list = <fun>
```

This is a common enough problem that `In_channel` has a function called `with_file` that automates this pattern:

```
# let load filename =
    In_channel.with_file filename ~f:(fun inc ->
      In_channel.input_lines inc |> List.map ~f:parse_line);;
val load : string -> float list list = <fun>
```

`In_channel.with_file` is built on top of `protect` so that it can clean up after itself in the presence of exceptions.

8.2.4 Catching Specific Exceptions

OCaml's exception-handling system allows you to tune your error-recovery logic to the particular error that was thrown. For example, `find_exn`, which we defined earlier in the chapter, throws `Key_not_found` when the element in question can't be found. Let's look at an example of how you could take advantage of this. In particular, consider the following function:

```
# let lookup_weight ~compute_weight alist key =
    try
      let data = find_exn alist key in
      compute_weight data
    with
    Key_not_found _ -> 0.;;
val lookup_weight :
  compute_weight:('a -> float) -> (string * 'a) list -> string ->
    float =
  <fun>
```

As you can see from the type, `lookup_weight` takes an association list, a key for looking up a corresponding value in that list, and a function for computing a floating-point weight from the looked-up value. If no value is found, then a weight of `0.` should be returned.

The use of exceptions in this code, however, presents some problems. In particular, what happens if `compute_weight` throws an exception? Ideally, `lookup_weight` should propagate that exception on, but if the exception happens to be `Key_not_found`, then that's not what will happen:

```
# lookup_weight ~compute_weight:(fun _ -> raise (Key_not_found "foo"))
  ["a",3; "b",4] "a";;
- : float = 0.
```

This kind of problem is hard to detect in advance because the type system doesn't tell you what exceptions a given function might throw. For this reason, it's usually best to avoid relying on the identity of the exception to determine the nature of a failure. A better approach is to narrow the scope of the exception handler, so that when it fires it's very clear what part of the code failed:

```
# let lookup_weight ~compute_weight alist key =
    match
      try Some (find_exn alist key)
      with _ -> None
    with
    | None -> 0.
    | Some data -> compute_weight data;;
val lookup_weight :
  compute_weight:('a -> float) -> (string * 'a) list -> string ->
    float =
  <fun>
```

This nesting of a `try` within a `match` expression is both awkward and involves some unnecessary computation (in particular, the allocation of the option). Happily, OCaml allows for exceptions to be caught by match expressions directly, which lets you write this more concisely as follows.

```
# let lookup_weight ~compute_weight alist key =
    match find_exn alist key with
    | exception _ -> 0.
    | data -> compute_weight data;;
val lookup_weight :
  compute_weight:('a -> float) -> (string * 'a) list -> string ->
    float =
  <fun>
```

Note that the exception keyword is used to mark the exception-handling cases.

Best of all is to avoid exceptions entirely, which we could do by using the exception-free function from Base, `List.Assoc.find`, instead:

```
# let lookup_weight ~compute_weight alist key =
    match List.Assoc.find ~equal:String.equal alist key with
    | None -> 0.
    | Some data -> compute_weight data;;
val lookup_weight :
  compute_weight:('a -> float) ->
  (string, 'a) Base.List.Assoc.t -> string -> float = <fun>
```

8.2.5 Backtraces

A big part of the value of exceptions is that they provide useful debugging information in the form of a stack backtrace. Consider the following simple program:

```
open Base
open Stdio
exception Empty_list

let list_max = function
  | [] -> raise Empty_list
  | hd :: tl -> List.fold tl ~init:hd ~f:(Int.max)

let () =
  printf "%d\n" (list_max [1;2;3]);
  printf "%d\n" (list_max [])
```

If we build and run this program, we'll get a stack backtrace that will provide some information about where the error occurred and the stack of function calls that were in place at the time of the error:

```
$ dune exec -- ./blow_up.exe
3
Fatal error: exception Dune__exe__Blow_up.Empty_list
Raised at Dune__exe__Blow_up.list_max in file "blow_up.ml", line 6,
    characters 10-26
Called from Dune__exe__Blow_up in file "blow_up.ml", line 11,
    characters 16-29
[2]
```

You can also capture a backtrace within your program by calling `Backtrace.Exn.most_recent`, which returns the backtrace of the most recently thrown exception. This is useful for reporting detailed information on errors that did not cause your program to fail.

This works well if you have backtraces enabled, but that isn't always the case. In fact, by default, OCaml has backtraces turned off, and even if you have them turned on at runtime, you can't get backtraces unless you have compiled with debugging symbols. Base reverses the default, so if you're linking in Base, you will have backtraces enabled by default.

Even using Base and compiling with debugging symbols, you can turn backtraces off via the OCAMLRUNPARAM environment variable, as shown below.

```
$ OCAMLRUNPARAM=b=0 dune exec -- ./blow_up.exe
3
Fatal error: exception Dune__exe__Blow_up.Empty_list
[2]
```

The resulting error message is considerably less informative. You can also turn backtraces off in your code by calling Backtrace.Exn.set_recording false.

There is a legitimate reason to run without backtraces: speed. OCaml's exceptions are fairly fast, but they're faster still if you disable backtraces. Here's a simple benchmark that shows the effect, using the core_bench package:

```
open Core
open Core_bench

exception Exit

let x = 0

type how_to_end = Ordinary | Raise | Raise_no_backtrace

let computation how_to_end =
  let x = 10 in
  let y = 40 in
  let _z = x + (y * y) in
  match how_to_end with
  | Ordinary -> ()
  | Raise -> raise Exit
  | Raise_no_backtrace -> raise_notrace Exit

let computation_with_handler how = try computation how with Exit -> ()

let () =
  [
    Bench.Test.create ~name:"simple computation" (fun () ->
        computation Ordinary);
    Bench.Test.create ~name:"computation w/handler" (fun () ->
        computation_with_handler Ordinary);
    Bench.Test.create ~name:"end with exn" (fun () ->
        computation_with_handler Raise);
    Bench.Test.create ~name:"end with exn notrace" (fun () ->
        computation_with_handler Raise_no_backtrace);
  ]
  |> Bench.make_command |> Command.run
```

We're testing four cases here:

- a simple computation with no exception,

- the same, but with an exception handler but no exception thrown,
- the same, but where an exception is thrown,
- and finally, the same, but where we throw an exception using `raise_notrace`, which is a version of `raise` which locally avoids the costs of keeping track of the backtrace.

Here are the results.

```
$ dune exec -- \
> ./exn_cost.exe -ascii -quota 1 -clear-columns time cycles
Estimated testing time 4s (4 benchmarks x 1s). Change using '-quota'.

  Name                      Time/Run   Cycls/Run
 ----------------------- ---------- -----------
  simple computation        1.84ns      3.66c
  computation w/handler     3.13ns      6.23c
  end with exn             27.96ns     55.69c
  end with exn notrace     11.69ns     23.28c
```

Note that we lose just a small number of cycles to setting up an exception handler, which means that an unused exception handler is quite cheap indeed. We lose a much bigger chunk, around 55 cycles, to actually raising an exception. If we explicitly raise an exception with no backtrace, it costs us about 25 cycles.

We can also disable backtraces, as we discussed, using `OCAMLRUNPARAM`. That changes the results a bit.

```
$ OCAMLRUNPARAM=b=0 dune exec -- \
> ./exn_cost.exe -ascii -quota 1 -clear-columns time cycles
Estimated testing time 4s (4 benchmarks x 1s). Change using '-quota'.

  Name                      Time/Run   Cycls/Run
 ----------------------- ---------- -----------
  simple computation        1.71ns      3.41c
  computation w/handler     3.04ns      6.05c
  end with exn             19.36ns     38.57c
  end with exn notrace     11.48ns     22.86c
```

The only significant change here is that raising an exception in the ordinary way becomes just a bit cheaper: 20 cycles instead of 55 cycles. But it's still not as fast as using `raise_notrace` explicitly.

Differences on this scale should only matter if you're using exceptions routinely as part of your flow control. That's not a common pattern, and when you do need it, it's better from a performance perspective to use `raise_notrace`. All of which is to say, you should almost always leave stack-traces on.

8.2.6 From Exceptions to Error-Aware Types and Back Again

Both exceptions and error-aware types are necessary parts of programming in OCaml. As such, you often need to move between these two worlds. Happily, Base comes with some useful helper functions to help you do just that. For example, given a piece of code that can throw an exception, you can capture that exception into an option as follows:

```
# let find alist key =
    Option.try_with (fun () -> find_exn alist key);;
val find : (string * 'a) list -> string -> 'a option = <fun>
# find ["a",1; "b",2] "c";;
- : int option = Base.Option.None
# find ["a",1; "b",2] "b";;
- : int option = Base.Option.Some 2
```

`Result` and `Or_error` have similar `try_with` functions. So, we could write:

```
# let find alist key =
    Or_error.try_with (fun () -> find_exn alist key);;
val find : (string * 'a) list -> string -> 'a Or_error.t = <fun>
# find ["a",1; "b",2] "c";;
- : int Or_error.t = Base__.Result.Error ("Key_not_found(\"c\")")
```

We can then reraise that exception:

```
# Or_error.ok_exn (find ["a",1; "b",2] "b");;
- : int = 2
# Or_error.ok_exn (find ["a",1; "b",2] "c");;
Exception: Key_not_found("c").
```

8.3 Choosing an Error-Handling Strategy

Given that OCaml supports both exceptions and error-aware return types, how do you choose between them? The key is to think about the trade-off between concision and explicitness.

Exceptions are more concise because they allow you to defer the job of error handling to some larger scope, and because they don't clutter up your types. But this concision comes at a cost: exceptions are all too easy to ignore. Error-aware return types, on the other hand, are fully manifest in your type definitions, making the errors that your code might generate explicit and impossible to ignore.

The right trade-off depends on your application. If you're writing a rough-and-ready program where getting it done quickly is key and failure is not that expensive, then using exceptions extensively may be the way to go. If, on the other hand, you're writing production software whose failure is costly, then you should probably lean in the direction of using error-aware return types.

To be clear, it doesn't make sense to avoid exceptions entirely. The maxim of "use exceptions for exceptional conditions" applies. If an error occurs sufficiently rarely, then throwing an exception is often the right behavior.

Also, for errors that are omnipresent, error-aware return types may be overkill. A good example is out-of-memory errors, which can occur anywhere, and so you'd need to use error-aware return types everywhere to capture those. Having every operation marked as one that might fail is no more explicit than having none of them marked.

In short, for errors that are a foreseeable and ordinary part of the execution of your production code and that are not omnipresent, error-aware return types are typically the right solution.