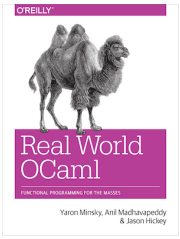

Buy in print and eBook.

Table of Contents

Chapter 7. 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 planning out 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.0 comments

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.0 comments

We'll start by describing the two basic approaches for reporting errors in OCaml: error-aware return types and exceptions.0 comments

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:0 comments

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

OCaml Utop ⁎ error-handling/main.topscript ⁎ all code

The option in the return type indicates that the function may not succeed in finding a suitable element:0 comments

```
# 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
```

OCaml Utop ⁎ error-handling/main.topscript , continued (part 1) ⁎ all code

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.0 comments

Consider the `compute_bounds` function. The function 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:0 comments

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

OCaml Utop ⁎ error-handling/main.topscript , continued (part 2) ⁎ all code

The `match` statement is used to handle the error cases, propagating a `None` in `hd` or `last` into the return value of `compute_bounds`.0 comments

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:0 comments

```
# 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, 'b) Hashtbl.t -> ('a, 'b) Hashtbl.t -> 'a list =
   <fun>
```

OCaml Utop ∗ error-handling/main.topscript , continued (part 3) ∗ all code

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.0 comments

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.0 comments

`Result.t` is meant to address this deficiency. The type is defined as follows:0 comments

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

OCaml ∗ error-handling/result.mli ∗ all code

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 promoted to the toplevel by `Core.Std`. As such, we can write:0 comments

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

OCaml Utop ∗ error-handling/main.topscript , continued (part 4) ∗ all code

without first opening the `Result` module.0 comments

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.0 comments

But which type to choose? Is it better to represent errors as strings? Some more structured representation like XML? Or something else entirely?0 comments

Core's answer to this question is the `Error.t` type, which tries to forge a good compromise between efficiency, convenience, and control over the presentation of errors.0 comments

It might not be obvious at first why efficiency is an issue at all. But generating error messages is an expensive business. An ASCII representation of a value can be quite time-consuming to construct, particularly if it includes expensive-to-convert numerical data.0 comments

`Error` gets around this issue through laziness. In particular, an `Error.t` allows you to put off generation of the error string until and unless you need it, which means a lot of the time you never have to construct it at all. You can of course construct an error directly from a string:0 comments

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

OCaml Utop ∗ error-handling/main.topscript , continued (part 5) ∗ all code

But you can also construct an `Error.t` from a *thunk*, i.e., a function that takes a single argument of type `unit`:0 comments

```
# Error.of_thunk (fun () ->
    sprintf "something went wrong: %f" 32.3343);;
 - : Error.t = something went wrong: 32.334300
```

OCaml Utop ∗ error-handling/main.topscript , continued (part 6) ∗ all code

In this case, we can benefit from the laziness of `Error`, since the thunk won't be called unless the `Error.t` is converted to a string.0 comments

The most common way to create `Error.t`s 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:0 comments

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

Scheme ∗ error-handling/sexpr.scm ∗ all code

S-expressions are supported by the Sexplib package that is distributed with Core and is the most common serialization format used in Core. Indeed, most types in Core come with built-in s-expression converters. Here's an example of creating an error using the sexp converter for times, `Time.sexp_of_t`:0 comments

```
# Error.create "Something failed a long time ago" Time.epoch Time.sexp_of_t;;
 - : Error.t =
 Something failed a long time ago: (1969-12-31 19:00:00.000000-05:00)
```

OCaml Utop ∗ error-handling/main.topscript , continued (part 7) ∗ all code

Note that the time isn't actually serialized into an s-expression until the error is printed out.0 comments

We're not restricted to doing this kind of error reporting with built-in types. This will be discussed in more detail in Chapter 17, *Data Serialization with S-Expressions*, but Sexplib comes with a language extension that can autogenerate sexp converters for newly generated types:0 comments

```
# let custom_to_sexp = <:sexp_of<float * string list * int>>;;
 val custom_to_sexp : float * string list * int -> Sexp.t = <fun>
# custom_to_sexp (3.5, ["a";"b";"c"], 6034);;
 - : Sexp.t = (3.5 (a b c) 6034)
```

OCaml Utop ∗ error-handling/main.topscript , continued (part 8) ∗ all code

We can use this same idiom for generating an error:0 comments

```
# Error.create "Something went terribly wrong"
    (3.5, ["a";"b";"c"], 6034)
    <:sexp_of<float * string list * int>> ;;
 - : Error.t = Something went terribly wrong: (3.5(a b c)6034)
```

OCaml Utop ∗ error-handling/main.topscript , continued (part 9) ∗ all code

`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:0 comments

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

OCaml Utop ∗ error-handling/main.topscript , continued (part 10) ∗ all code

The type `'a Or_error.t` is just a shorthand for `('a,Error.t) Result.t`, and it is, after `option`, the most common way of returning errors in Core.0 comments

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:0 comments

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

OCaml Utop ∗ error-handling/main.topscript , continued (part 11) ∗ all code

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 `bind`s:0 comments

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

OCaml Utop ⋆ error-handling/main.topscript , continued (part 12) ⋆ all code

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 18, *Concurrent Programming with Async*:0 comments

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

OCaml Utop ⋆ error-handling/main.topscript , continued (part 13) ⋆ all code

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.0 comments

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:0 comments

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

OCaml Utop ⋆ error-handling/main.topscript , continued (part 14) ⋆ all code

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.0 comments

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.0 comments

You can trigger an exception by, for example, dividing an integer by zero:0 comments

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

OCaml Utop ⋆ error-handling/main.topscript , continued (part 15) ⋆ all code

And an exception can terminate a computation even if it happens nested somewhere deep within it:0 comments

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

OCaml Utop ⋆ error-handling/main.topscript , continued (part 16) ⋆ all code

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:0 comments

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


 1
 3
 0
 Exception: Division_by_zero.
```

OCaml Utop ⋆ error-handling/main.topscript , continued (part 17) ⋆ all code

In addition to built-in exceptions like `Divide_by_zero`, OCaml lets you define your own:0 comments

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

OCaml Utop ⋆ error-handling/main.topscript , continued (part 18) ⋆ all code

Exceptions are ordinary values and can be manipulated just like other OCaml values:0 comments

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

OCaml Utop ⋆ error-handling/main.topscript , continued (part 19) ⋆ all code

Exceptions are all of the same type, `exn`. The `exn` type is something of a special case in the OCaml type system. It is similar to the variant types we encountered in Chapter 6, *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.0 comments

The following function uses the `Key_not_found` exception we defined above to signal an error:0 comments

```
# let rec find_exn alist key = match alist with
    | [] -> raise (Key_not_found key)
    | (key',data) :: tl -> if 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").
```

OCaml Utop ⋆ error-handling/main.topscript , continued (part 20) ⋆ all code

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 Core.0 comments

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:0 comments

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

OCaml Utop ⋆ error-handling/main.topscript , continued (part 21) ⋆ all code

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:0 comments

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

OCaml Utop ⁎ error-handling/main.topscript , continued (part 22) ⁎ all code

`forever` doesn't return a value for a different reason: it's an infinite loop.0 comments

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.0 comments

Declaring Exceptions Using with sexp

OCaml can't always generate a useful textual representation of an exception. For example:0 comments

```
# exception Wrong_date of Date.t;;
 exception Wrong_date of Date.t
# Wrong_date (Date.of_string "2011-02-23");;
 - : exn = Wrong_date(_)
```

OCaml Utop ⁎ error-handling/main.topscript , continued (part 23) ⁎ all code

But if we declare the exception using `with sexp` (and the constituent types have sexp converters), we'll get something with more information:0 comments

```
# exception Wrong_date of Date.t with sexp;;
 exception Wrong_date of Date.t
# Wrong_date (Date.of_string "2011-02-23");;
 - : exn = (//toplevel//.Wrong_date 2011-02-23)
```

OCaml Utop ⁎ error-handling/main.topscript , continued (part 24) ⁎ all code

The period in front of `Wrong_date` is there because the representation generated by `with 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 toplevel, rather than in a module.0 comments

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 17, *Data Serialization with S-Expressions*.0 comments

Helper Functions for Throwing Exceptions

OCaml and Core provide a number of helper functions to simplify the task of throwing exceptions. The simplest one is `failwith`, which could be defined as follows:0 comments

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

OCaml Utop ⁎ error-handling/main.topscript , continued (part 25) ⁎ all code

There are several other useful functions for raising exceptions, which can be found in the API documentation for the `Common` and `Exn` modules in Core.0 comments

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:0 comments

```
# 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
```

OCaml Utop ⋆ error-handling/main.topscript , continued (part 26) ⋆ all code

Here we use `assert false`, which means that the `assert` is guaranteed to trigger. In general, one can put an arbitrary condition in the assertion.0 comments

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`:0 comments

```
# 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// 5 13).
```

OCaml Utop ⋆ error-handling/main.topscript , continued (part 27) ⋆ all code

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.0 comments

Exception Handlers

So far, we've only seen exceptions fully terminate the execution of a computation. But often, we want a program to be able to respond to and recover from an exception. This is achieved through the use of *exception handlers*.0 comments

In OCaml, an exception handler is declared using a `try/with` statement. Here's the basic syntax.0 comments

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

Syntax ⋆ error-handling/try_with.syntax ⋆ all code

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.0 comments

But if the evaluation of the body throws an exception, then the exception will be fed to the pattern-match statements 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 righthand side of the matching pattern.0 comments

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.0 comments

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 reminders, formatted as s-expressions:0 comments

```
# let reminders_of_sexp =
    <:of_sexp<(Time.t * string) list>>
  ;;
val reminders_of_sexp : Sexp.t -> (Time.t * string) list = <fun>
# let load_reminders filename =
    let inc = In_channel.create filename in
    let reminders = reminders_of_sexp (Sexp.input_sexp inc) in
```

```
    In_channel.close inc;
    reminders
  ;;
val load_reminders : string -> (Time.t * string) list = <fun>
```

OCaml Utop ⋆ error-handling/main.topscript , continued (part 28) ⋆ all code

The problem with this code is that the function that loads the s-expression and parses it into a list of `Time.t`/`string` pairs might 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.0 comments

We can fix this using Core's `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 `load_reminders`:0 comments

```
# let load_reminders filename =
    let inc = In_channel.create filename in
    protect ~f:(fun () -> reminders_of_sexp (Sexp.input_sexp inc))
      ~finally:(fun () -> In_channel.close inc)
  ;;
val load_reminders : string -> (Time.t * string) list = <fun>
```

OCaml Utop ⋆ error-handling/main.topscript , continued (part 29) ⋆ all code

This is a common enough problem that `In_channel` has a function called `with_file` that automates this pattern:0 comments

```
# let reminders_of_sexp filename =
    In_channel.with_file filename ~f:(fun inc ->
      reminders_of_sexp (Sexp.input_sexp inc))
  ;;
val reminders_of_sexp : string -> (Time.t * string) list = <fun>
```

OCaml Utop ⋆ error-handling/main.topscript , continued (part 30) ⋆ all code

`In_channel.with_file` is built on top of `protect` so that it can clean up after itself in the presence of exceptions.0 comments

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, `List.find_exn` throws `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:0 comments

```
# let lookup_weight ~compute_weight alist key =
    try
      let data = List.Assoc.find_exn alist key in
      compute_weight data
    with
      Not_found -> 0. ;;
val lookup_weight :
  compute_weight:('a -> float) -> ('b, 'a) List.Assoc.t -> 'b -> float =
  <fun>
```

OCaml Utop ⋆ error-handling/main.topscript , continued (part 31) ⋆ all code

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.0 comments

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 `Not_found`, then that's not what will happen:0 comments

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

OCaml Utop ⋆ error-handling/main.topscript , continued (part 32) ⋆ all code

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 generally better 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:0 comments

```
# let lookup_weight ~compute_weight alist key =
    match
      try Some (List.Assoc.find_exn alist key)
      with _ -> None
    with
    | None -> 0.
    | Some data -> compute_weight data ;;
 val lookup_weight :
   compute_weight:('a -> float) -> ('b, 'a) List.Assoc.t -> 'b -> float =
   <fun>
```

OCaml Utop ∗ error-handling/main.topscript , continued (part 33) ∗ all code

At this point, it makes sense to simply use the nonexception-throwing function, `List.Assoc.find`, instead:0 comments

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

OCaml Utop ∗ error-handling/main.topscript , continued (part 34) ∗ all code

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:0 comments

```
open Core.Std
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 [])
```

OCaml ∗ error-handling/blow_up.ml ∗ all code

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:0 comments

```
$ corebuild blow_up.byte
$ ./blow_up.byte
 3
 Fatal error: exception Blow_up.Empty_list
 Raised at file "blow_up.ml", line 5, characters 16-26
 Called from file "blow_up.ml", line 10, characters 17-28
```

Terminal ∗ error-handling/build_blow_up.out ∗ all code

You can also capture a backtrace within your program by calling `Exn.backtrace`, 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.0 comments

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. Core reverses the default, so if you're linking in Core, you will have backtraces enabled by default.0 comments

Even using Core and compiling with debugging symbols, you can turn backtraces off by setting the `OCAMLRUNPARAM` environment variable to be empty:0 comments

```
$ corebuild blow_up.byte
$ OCAMLRUNPARAM= ./blow_up.byte
 3
```

```
 Fatal error: exception Blow_up.Empty_list
```

Terminal ∗ error-handling/build_blow_up_notrace.out ∗ all code

The resulting error message is considerably less informative. You can also turn backtraces off in your code by calling `Backtrace.Exn.set_recording false`.0 comments

There is a legitimate reasons to run without backtraces: speed. OCaml's exceptions are fairly fast, but they're even faster still if you disable backtraces. Here's a simple benchmark that shows the effect, using the `core_bench` package:0 comments

```
open Core.Std
open Core_bench.Std

let simple_computation () =
  List.range 0 10
  |> List.fold ~init:0 ~f:(fun sum x -> sum + x * x)
  |> ignore

let simple_with_handler () =
  try simple_computation () with Exit -> ()

let end_with_exn () =
  try
    simple_computation ();
    raise Exit
  with Exit -> ()

let () =
  [ Bench.Test.create ~name:"simple computation"
      (fun () -> simple_computation ());
    Bench.Test.create ~name:"simple computation w/handler"
      (fun () -> simple_with_handler ());
    Bench.Test.create ~name:"end with exn"
      (fun () -> end_with_exn ());
  ]
  |> Bench.make_command
  |> Command.run
```

OCaml ∗ error-handling/exn_cost.ml ∗ all code

We're testing three cases here: a simple computation with no exceptions; the same computation with an exception handler but no thrown exceptions; and finally the same computation where we use the exception to do the control flow back to the caller.0 comments

If we run this with stacktraces on, the benchmark results look like this:0 comments

```
$ corebuild -pkg core_bench exn_cost.native
$ ./exn_cost.native -ascii cycles
 Estimated testing time 30s (change using -quota SECS).

  Name                          Time/Run   Cycles/Run   % of max
 ----------------------------- ---------- ------------ ----------
  simple computation               74.43          171      71.63
  simple computation w/handler     92.46          213      88.98
  end with exn                       104          239     100.00
```

Terminal ∗ error-handling/run_exn_cost.out ∗ all code

Here, we see that we lose something like 30 cycles to adding an exception handler, and 60 more to actually throwing and catching an exception. If we turn backtraces off, then the results look like this:0 comments

```
$ OCAMLRUNPARAM= ./exn_cost.native -ascii cycles
 Estimated testing time 30s (change using -quota SECS).

  Name                          Time/Run   Cycles/Run   % of max
 ----------------------------- ---------- ------------ ----------
  simple computation               84.85          195      82.06
  simple computation w/handler     91.95          211      88.93
  end with exn                       103          238     100.00
```

Terminal ∗ error-handling/run_exn_cost_notrace.out ∗ all code

Here, the handler costs about the same, but the exception itself costs only 25, as opposed to 60

additional cycles. All told, this should only matter if you're using exceptions routinely as part of your flow control, which is in most cases a stylistic mistake anyway.0 comments

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, Core 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:0 comments

```
# 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 = None
# find ["a",1; "b",2] "b";;
 - : int option = Some 2
```

OCaml Utop ⋆ error-handling/main.topscript , continued (part 35) ⋆ all code

And `Result` and `Or_error` have similar `try_with` functions. So, we could write:0 comments

```
# 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 = Core_kernel.Result.Error ("Key_not_found(\"c\")")
```

OCaml Utop ⋆ error-handling/main.topscript , continued (part 36) ⋆ all code

And then we can reraise that exception:0 comments

```
# 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\")").
```

OCaml Utop ⋆ error-handling/main.topscript , continued (part 37) ⋆ all code

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.0 comments

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.0 comments

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.0 comments

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.0 comments

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.0 comments

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.0 comments