10 Handling anomalous conditions

Despite best efforts, on occasion a condition arises – let's call it an ANOMALY – that a function can't handle. What to do? In this chapter, we present two approaches. The function can return a value that indicates the anomaly, thereby handling the anomaly explicitly. Alternatively, the function can stop normal execution altogether, throwing control to some handler of the anomaly. In OCaml, the first approach involves *option types*, the second *exceptions*.

As a concrete example, consider a function to calculate the MEDIAN number in a list of integer values, that is, the value that has an equal number of smaller and larger values. The median can be calculated by sorting all of the values in the list and taking the middle element of the sorted list. Taking advantage of a few functions from the `List` module (`sort`, `length`, and `nth`) and the `Stdlib` module (`compare`),[1] we can define

```
# let median (lst: 'a list) : 'a =
#   nth (sort compare lst) (length lst / 2) ;;
val median : 'a list -> 'a = <fun>
```

We can test it out on a few lists:

```
# median [1; 5; 9; 7; 3] ;;
- : int = 5
# median [1; 2; 3; 4; 3; 2; 1] ;;
- : int = 2
# median [1; 1; 1; 1; 1] ;;
- : int = 1
# median [7] ;;
- : int = 7
```

The function works fine most of the time, but there is one anomalous condition to consider, where the median isn't well defined: What should the `median` function do on the empty list?

[1] Since we make heavy use of the `List` module functions in this chapter, we will open the module (but preserve `compare` as the `Stdlib` version)

```
# open List ;;
# let compare = Stdlib.compare ;;
val compare : 'a -> 'a -> int = <fun>
```

so as to avoid having to prefix each use of the functions with the `List.` module qualifier. The issue will become clearer when modules are fully introduced in Chapter 12.

10.1 A non-solution: Error values

You might have thought to return a special ERROR VALUE in the anomalous case. Perhaps `0` or `-1` or `MAX_INT` come to mind as possible error values. Augmenting the code to return a globally defined error value might look like this:

```
# let cERROR = -1 ;;
val cERROR : int = -1

# let median (lst: 'a list) : 'a =
#   if lst = [] then cERROR
#   else nth (sort compare lst) (length lst / 2) ;;
val median : int list -> int = <fun>
```

There are two problems. First, the method can lead to gratuitous type instantiation; second, and more critically, it manifests in-band signaling.

Check the types inferred for the two versions of `median` above. The original is appropriately polymorphic, of type `'a list -> 'a`. But because the error value `cERROR` used in the second version is of type `int`, `median` becomes instantiated to `int list -> int`. The code no longer applies outside the type of the error value, restricting its generality and utility. And there is a deeper problem.

Consider the sad fate of poor Christopher Null, a technology journalist with a rather inopportune name. Apparently, there is a fair amount of software that uses the string `"null"` as an error value for cases in which no last name was provided. Errors can then be checked for using code like

```
if last_name = "null" then ...
```

You see the problem. Poor Mr. Null reports that

> I've been embroiled in a cordial email battle with Bank of America, literally for years, over my email address, which is simply null@nullmedia.com. Using null as a mailbox name simply does not work at B of A. The system will not accept it, period. (Null, 2015)

These kinds of problems confront poor Mr. Null on a regular basis.

Null has fallen afoul of IN-BAND SIGNALING of errors, in which an otherwise valid value is used to indicate an error. The string `"null"` is, of course, a valid string that, for all the programmer knows, might be someone's name, yet it is used to indicate a failure condition in which no name was provided. (The solution is not to use a string, `"dpfnzzlwrpf"` say,[2] that is less likely to be someone's last name as the error value. That merely postpones the problem.)

Similarly, `0` or `-1` or `MAX_INT` are all possible values for the median of an integer list. Using one of them as an in-band error value means

[2] In fact, "Dpfnzzlwrpf" is the name of a fictitious corporation in Jonathan Caws-Elwitt's "Letter to a Customer". (Conley, 2009) Could it also be a last name? Why not? For a while, it was my username on Skype. True story.

that users of the `median` function can't tell the difference between the value being the true median or the median being undefined.

Having dismissed the in-band error signaling approach, we turn to better solutions.

10.2 Option types

The first approach, like the in-band error value approach, still handles the problem explicitly, right in the return value of the function. However, rather than returning an in-band value, an `int` (or whatever the type of the list elements is), the function will return an out-of-band `None` value, that has been added to the `int` type to form an *optional* `int`, a value of type `int option`.

Option types are another kind of structured type, beyond the lists, tuples, and records from Chapter 7. The postfix type constructor `option` creates an option type from a base type, just as the postfix type constructor `list` does. There are two value constructors for option type values: `None` (connoting an anomalous value), and the prefix value constructor `Some`. The argument to `Some` is a value of the base type.

For the median function, we'll use an `int option` as the return value, or, more generically, an `'a option`. In the anomalous condition, we return `None`, and in the normal condition in which a well-defined median v can be computed, we return `Some` v.

```
# let median (lst: 'a list) : 'a option =
#   if lst = [] then None
#   else Some (nth (sort compare lst) (length lst / 2)) ;;
val median : 'a list -> 'a option = <fun>

# median [1; 2; 3; 4; 42] ;;
- : int option = Some 3
# median [] ;;
- : 'a option = None
```

This version of the `median` function when applied to an `int list` does not return an `int`, even when the median is well defined. It returns an `int option`, which is a distinct type altogether. Nonetheless, a caller of this function might want access to the `int` wrapped inside the `int option` value. As with all structured types, we access the component elements of an option value via pattern matching, as in this example function, which replicates the (deprecated) in-band value solution, returning the median of the list, or the error value if no median exists:

```
# let median_or_error (lst : int list) : int =
#   match median lst with
#   | None -> cERROR
#   | Some v -> v ;;
val median_or_error : int list -> int = <fun>
```

In implementing `median` above, we used the polymorphic function `nth : 'a list -> int -> 'a` provided by the `List` module, which given a list `lst` and an integer `index` returns the element of `lst` at the given `index` (numbered starting with 0).

```
# List.nth [1; 2; 4; 8] 2 ;;
- : int = 4
# List.nth [true; false; false] 0 ;;
- : bool = true
```

Exercise 74

Why do you think `nth` was designed so as to take its list argument before its index argument? The designers expected that this would be a more commonly needed abstraction than a function that returns the n-th element of a list for a particular n.

If we were to reimplement this function, it might look something like this:

```
# let rec nth (lst : 'a list) (n : int) : 'a =
#   match lst with
#   | hd :: tl ->
#       if n = 0 then hd
#       else nth tl (n - 1) ;;
Lines 2-5, characters 0-19:
2 | match lst with
3 | | hd :: tl ->
4 | if n = 0 then hd
5 | else nth tl (n - 1)...
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]
val nth : 'a list -> int -> 'a = <fun>
```

This definition works, as shown in the following examples:

```
# nth [1; 2; 3] 1 ;;
- : int = 2
# nth [0; 1; 2] (nth [1; 2; 3] 1) ;;
- : int = 2
```

However, OCaml has warned us that the pattern match in the definition of `nth` is not exhaustive – there are possible values that will match none of the provided patterns – and helpfully provides the missing case, the empty list. Of course, if we ask to take the n-th element of an empty list, there is no element to take; this represents an anomalous condition.

Leaving the handling of this case implicit violates the edict of intention; we should clearly express what happens in all cases. Once again, we can use option types to explicitly mark the condition in the return value. We do so in a function called `nth_opt`.[3]

[3] We use the suffix `_opt` to mark functions that return an optional value, as is conventional in OCaml library functions. In fact, as noted below, the `List` module provides an `nth_opt` function in addition to its `nth` function.

```
# let rec nth_opt (lst : 'a list) (n : int) : 'a option =
#   match lst with
#   | [] -> None
#   | hd :: tl ->
#       if n = 0 then Some hd
#       else nth_opt tl (n - 1) ;;
val nth_opt : 'a list -> int -> 'a option = <fun>

# nth_opt [1; 2; 3] 1 ;;
- : int option = Some 2
# nth_opt [1; 2; 3] 5 ;;
- : int option = None
```

Exercise 75

Another anomalous condition for `nth` and `nth_opt` is the use of a negative index. What currently is the behavior of `nth_opt` with negative indices? Revise the definition of `nth_opt` to appropriately handle this case as well.

Exercise 76

Define a function `last_opt : ’a list -> ’a option` that returns the last element in a list (as an element of the option type) if there is one, and `None` otherwise.

```
# last_opt [] ;;
- : 'a option = None
# last_opt [1; 2; 3; 4; 5] ;;
- : int option = Some 5
```

Exercise 77

The variance of a sequence of n numbers $x_1, \ldots, x_n$ is given by the following equation:

$$\frac{\sum_{i=1}^{n}(x_i - m)^2}{n-1}$$

where n is the number of elements in the sequence, m is the arithmetic mean (or average) of the elements in the sequence, and x_i is the i-th element in the sequence. The variance is only well defined for sequences with two or more elements. (Do you see why?)

Define a function `variance : float list -> float option` that returns `None` if the list has fewer than two elements. Otherwise, it should return the variance of the numbers in its list argument, wrapped appropriately for its return type.[4] For example:

```
# variance [1.0; 2.0; 3.0; 4.0; 5.0] ;;
- : float option = Some 2.5
# variance [1.0] ;;
- : float option = None
```

[4] If you want to compare your output with an online calculator, make sure you find one that calculates the (unbiased) sample variance.

Remember to use the floating point version of the arithmetic operators when operating on floats (`+.`, `*.`, etc). The function `float` can convert (“cast”) an `int` to a `float`.

10.2.1 Option poisoning

There is a problem with using option types to handle anomalies, as in `nth_opt`. Whenever we want to use the value of an `nth_opt` element in a further computation, we need to carefully extract the value from the option type. We can’t, for instance, merely write

```
# nth_opt [0; 1; 2] (nth_opt [1; 2; 3] 1) ;;
Line 1, characters 18-39:
1 | nth_opt [0; 1; 2] (nth_opt [1; 2; 3] 1) ;;
```

```
                        ^^^^^^^^^^^^^^^^^^^^^
Error: This expression has type int option
       but an expression was expected of type int
```

Instead we must work inside out, painstakingly extracting values and passing on `None`s:

```
# match (nth_opt [1; 2; 3] 1) with
# | None -> None
# | Some v -> nth_opt [0; 1; 2] v ;;
- : int option = Some 2
```

And if that result is part of a further computation, even something as simple as adding 1 to it, we have to resort to

```
# match (nth_opt [1; 2; 3] 1) with
# | None -> None
# | Some v ->
#     match nth_opt [0; 1; 2] v with
#     | None -> None
#     | Some v -> Some (v + 1) ;;
- : int option = Some 3
```

Much of the elegance of the functional programming paradigm, the ability to simply embed function applications with other functional applications, is lost. We call this phenomenon OPTION POISONING: The introduction of an option type in an embedded computation requires verbose extraction of values and reinjecting them into option types as the computation continues. (Option poisoning is a particular instance of the dreaded programming phenomenon of the PYRAMID OF DOOM.)

Functions that regularly display anomalous conditions that ought to be directly handled by the caller are well suited for use of option types. But where an anomalous condition is rare and isn't the kind of thing that the caller should handle, an alternative approach is useful, to avoid the pyramid of doom. Rather than explicitly marking the occurrence of an anomaly in the return value, it can be implicitly dealt with by changing the flow of control of the program entirely. This is the approach based on exceptions, to which we now turn.[5]

[5] Newer techniques, such as OPTIONAL CHAINING in the Swift programming language, deal with option poisoning in a more elegant way, providing a middle ground between the verbose option handling of OCaml and the use of exceptions. For the programming-language-theory-inclined, the MONAD concept from category theory, first imported into programming languages with Haskell, generalizes the concept.
The lesson here is that continuing progress is being made in the design of programming languages to deal with new and recurring programming issues.

10.3 Exceptions

Instead of modifying the return type of `nth` to allow for returning a `None` marker of an anomaly, we can leave the return type unchanged, and in case of anomaly, raise an EXCEPTION.

When an exception is raised, execution of the function *stops*. Of course, if execution stops, the function can't return a value, which is appropriate given that the existence of the anomaly means that *there's no appropriate value to return.*

What about the function that called the one that raised the exception? It is expecting a value of a certain type to be returned, but in this case, no such value is supplied. The calling function thus can't return either. It stops too. And so on and so forth.

We can write a version of `nth` that raises an exception when the index is too large.

```
# let rec nth (lst : 'a list) (n : int) : 'a =
#   match lst with
#   | [] -> raise Exit
#   | hd :: tl ->
#       if n = 0 then hd
#       else nth tl (n - 1) ;;
val nth : 'a list -> int -> 'a = <fun>

# nth [1; 2; 3] 1 ;;
- : int = 2
# nth [1; 2; 3] 5 ;;
Exception: Stdlib.Exit.
# (nth [0; 1; 2] (nth [1; 2; 3] 1)) + 1 ;;
- : int = 3
```

There are several things to notice here. First, the return type of `nth` remains `'a`, not `'a option`. Under normal conditions, it returns the n-th element itself, not an option-wrapped version thereof. This allows its use in embedded applications (as in the third example above) without leading to the dreaded option poisoning. When an error does occur, as in the second example, execution stops and a message is printed by the OCaml REPL ("`Exception: Stdlib.Exit.`") describing the exception that was raised, namely, the `Exit` exception defined in the `Stdlib` library module. No value is returned from the computation at all, so no value is ever printed by the REPL.

The code that actually raises the `Exit` exception is in the third line of `nth`: `raise Exit`. The built-in `raise` function takes as argument an expression of type `exn`, the type for exceptions. As it turns out, `Exit` is a value of that type, as can be verified directly:

```
# Exit ;;
- : exn = Stdlib.Exit
```

The `Exit` exception is provided in the `Stdlib` module as a kind of catch-all exception, but other exceptions are more appropriate to raise in different circumstances.

- The value constructor `Invalid_argument : string -> exn`, is intended for use when an argument to a function is inappropriate. It would be appropriate to use when the index of `nth` is negative.

- The value constructor `Failure : string -> exn`, is intended for use when a function isn't well-defined as called. It would be appropriate to use when the index of `nth` is too large for the given list.

Both of these constructors take a string argument, typically used to provide an explanation of what went wrong. The explanation can be used when the exception information is handled, for instance, by the REPL printing its error message.

Taking advantage of these exceptions, `nth` can be rewritten as

```
# let rec nth (lst : 'a list) (n : int) : 'a =
#   if n < 0 then
#     raise (Invalid_argument "nth: negative index")
#   else
#     match lst with
#     | [] -> raise (Failure "nth: index too large")
#     | hd :: tl ->
#         if n = 0 then hd
#         else nth tl (n - 1) ;;
val nth : 'a list -> int -> 'a = <fun>
```

```
# nth [1; 2; 4; 8] ~-3 ;;
Exception: Invalid_argument "nth: negative index".
# nth [1; 2; 4; 8] 1 ;;
- : int = 2
# nth [1; 2; 4; 8] 42 ;;
Exception: Failure "nth: index too large".
```

We've dealt with both of the anomalous conditions by raising appropriate exceptions.

Not coincidentally, the `List.nth` function (in the `List` library module) works exactly this way, raising `Invalid_argument` and `Failure` exceptions under just these circumstances. But a `List.nth_opt` function is also provided, for cases in which the explicit marking of anomalies with an `option` type is more appropriate.

Returning to the `median` example above, and repeated here for reference (but this time using our own implementation of `nth`),

```
# let median (lst : 'a list) : 'a =
#   nth (sort compare lst) (length lst / 2) ;;
val median : 'a list -> 'a = <fun>
```

this code doesn't use option types and doesn't use the `raise` function to raise any exceptions. What *does* happen when the anomalous condition occurs?

```
# median [] ;;
Exception: Failure "nth: index too large".
```

An exception was raised, not by the `median` function, but by our `nth` function that it calls, which raises a `Failure` exception when it is called to take an element of the empty list. The exception propagates from the `nth` call to the `median` call to the top level of the REPL.

10.3.1 Handling exceptions

Perhaps you, as the writer of some code, have an idea about how to handle particular anomalies that might otherwise raise an exception. Rather than allow the exception to propagate to the top level, you might want to handle the exception yourself. The `try` ⟨⟩ `with` ⟨⟩ construct allows for this.

The syntax of the construction is

$$\begin{aligned}\langle \mathit{expr}\rangle ::= {} & \texttt{try}\ \langle \mathit{expr}_{\mathit{value}}\rangle\ \texttt{with} \\ & \mid \langle \mathit{exnpattern}_1\rangle\ \texttt{->}\ \langle \mathit{expr}_1\rangle \\ & \mid \langle \mathit{exnpattern}_2\rangle\ \texttt{->}\ \langle \mathit{expr}_2\rangle \\ & \ldots\end{aligned}$$

where $\langle \mathit{expr}_{\mathit{value}}\rangle$ is an expression that may raise an exception, and the $\langle \mathit{exnpattern}_i\rangle$ are patterns that match against OCaml exception expressions, rather than the normal algebraic data structures.

If the $\langle \mathit{expr}_{\mathit{value}}\rangle$ evaluates without exception, its value is returned. However, if its evaluation raises an exception, that exception is pattern-matched sequentially against the $\langle \mathit{exnpattern}_i\rangle$ much as in a `match` expression; for the first such pattern that matches, the corresponding $\langle \mathit{expr}_i\rangle$ is evaluated and its value returned from the `try`.

For example, we can implement `nth_opt` in terms of `nth` by embedding the call to `nth` within a `try` ⟨⟩ `with` ⟨⟩ :[6]

```
# let nth_opt (lst : 'a list) (index : int) : 'a option =
#   try
#     Some (nth lst index)
#   with
#   | Failure _
#   | Invalid_argument _ -> None ;;
val nth_opt : 'a list -> int -> 'a option = <fun>

# nth_opt [1; 2; 3] 0 ;;
- : int option = Some 1
# nth_opt [1; 2; 3] (-1) ;;
- : int option = None
# nth_opt [1; 2; 3] 4 ;;
- : int option = None
```

[6] We've taken advantage of the ability to use the same result expression for multiple patterns, as described in Section 7.2.1.

This implementation of `nth_opt` attempts to evaluate `Some (nth lst index)`. Under normal conditions, the call to `nth` returns a value v, in which case `Some` v is the result of the `try` and of the function itself. But if an exception is raised in the evaluation of the `try` – presumably by an anomalous condition in the call to `nth` – the exception raised will be matched against the two patterns and the result of that pattern match will be used. If `nth` raises either a `Failure` exception or an `Invalid_argument` exception, the result of the `try...with` will be

`None` (as is appropriate for an implementation of `nth_opt`). If any other exception is raised, no pattern will match and the exception will continue to propagate.

10.3.2 Zipping lists

As another example of handling anomalous conditions, we consider a function for “zipping” lists. The result of zipping two lists together is a list of corresponding pairs of elements of the original lists. A `zip` function in OCaml ought to have the following behavior:

```
# zip ['a'; 'b'; 'c']
#     [ 1 ;  2 ;  3 ] ;;
- : (char * int) list = [('a', 1); ('b', 2); ('c', 3)]
```

Let’s try to define the function, starting with its type. The `zip` function takes two lists, with types, say, `'a list` and `'b list`, and returns a list of pairs each of which has an element from the first list (of type `'a`) and an element from the second (of type `'b`). The pairs are thus of type `'a * 'b` and the return value of type `('a * 'b) list`. The type of the whole function, then, is `'a list -> 'b list -> ('a * 'b) list`. From this, the header follows directly.

```
let rec zip (xs : 'a list)
            (ys : 'b list)
          : ('a * 'b) list =
  ...
```

We’ll need the first elements of each of the lists, so we match on both lists (as a pair) to extract their parts

```
let rec zip (xs : 'a list)
            (ys : 'b list)
          : ('a * 'b) list =
  match xs, ys with
  | [], [] -> ...
  | xhd :: xtl, yhd :: ytl -> ...
```

If the lists are empty, the list of pairs of their elements is empty too.

```
let rec zip (xs : 'a list)
            (ys : 'b list)
          : ('a * 'b) list =
  match xs, ys with
  | [], [] -> []
  | xhd :: xtl, yhd :: ytl -> ...
```

Otherwise, the zip of the non-empty lists starts with the two heads paired. The remaining elements are the zip of the tails.

```
let rec zip (xs : 'a list)
            (ys : 'b list)
```

```
          : ('a * 'b) list =
  match xs, ys with
  | [], [] -> []
  | xhd :: xtl, yhd :: ytl ->
      (xhd, yhd) :: (zip xtl ytl) ;;
```

You'll notice that there's an issue. And if you don't notice, the interpreter will, as soon as we enter this definition:

```
# let rec zip (xs : 'a list)
#             (ys : 'b list)
#           : ('a * 'b) list =
#   match xs, ys with
#   | [], [] -> []
#   | xhd :: xtl, yhd :: ytl ->
#       (xhd, yhd) :: (zip xtl ytl) ;;
Lines 4-7, characters 0-27:
4 | match xs, ys with
5 | | [], [] -> []
6 | | xhd :: xtl, yhd :: ytl ->
7 | (xhd, yhd) :: (zip xtl ytl)...
Warning 8 [partial-match]: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
([], _::_)
val zip : 'a list -> 'b list -> ('a * 'b) list = <fun>
```

There are missing match cases, in particular, when one of the lists is empty and the other isn't. This can arise whenever the two lists are of different lengths. In such a case, the zip of two lists is not well defined.

As usual, we have two approaches to addressing the anomaly, with options and with exceptions. We'll pursue them in order.

We can make explicit the possibility of error values by returning an option type.

```
let rec zip_opt (xs : 'a list)
                (ys : 'b list)
              : ('a * 'b) list option = ...
```

The normal match cases can return their corresponding option type value using the `Some` constructor.

```
let rec zip_opt (xs : 'a list)
                (ys : 'b list)
              : ('a * 'b) list option =
  match xs, ys with
  | [], [] -> Some []
  | xhd :: xtl, yhd :: ytl ->
      Some ((xhd, yhd) :: (zip_opt xtl ytl)) ;;
```

Finally, we can add a wild-card match pattern for the remaining cases.

```
# let rec zip_opt (xs : 'a list)
#                 (ys : 'b list)
#               : ('a * 'b) list option =
#   match xs, ys with
#   | [], [] -> Some []
#   | xhd :: xtl, yhd :: ytl ->
#       Some ((xhd, yhd) :: (zip_opt xtl ytl))
#   | _, _ -> None ;;
Line 7, characters 20-37:
7 | Some ((xhd, yhd) :: (zip_opt xtl ytl))
                        ^^^^^^^^^^^^^^^^^
Error: This expression has type ('c * 'd) list option
       but an expression was expected of type ('a * 'b) list
```

The interpreter tells us that there's a type problem. The recursive call `zip_opt xtl ytl` is of type `('c * 'd) list option` but the cons requires an `('a * 'b) list`. What we have here is a bad case of option poisoning. We'll have to decompose the return value of the recursive call to extract the list within, handling the `None` case at the same time.

```
# let rec zip_opt (xs : 'a list)
#                 (ys : 'b list)
#               : ('a * 'b) list option =
#   match xs, ys with
#   | [], [] -> Some []
#   | xhd :: xtl, yhd :: ytl ->
#       match zip_opt xtl ytl with
#       | None -> None
#       | Some ztl -> Some ((xhd, yhd) :: ztl)
#   | _, _ -> None ;;
Line 10, characters 2-6:
10 | | _, _ -> None ;;
       ^^^^
Error: This pattern matches values of type 'a * 'b
       but a pattern was expected which matches values of type
         ('c * 'd) list option
```

Now what? The interpreter complains of another type mismatch, this time in the final pattern, which is of type `'a * 'b`, but which, for some reason, the interpreter thinks should be of type `('c * 'd) list option`. This kind of error is one of the most confusing for beginning OCaml programmers.

Exercise 78
Try to see if you can diagnose the problem before reading on.

The indentation of this code notwithstanding, the final pattern match is associated with the *inner* `match`, not the *outer* one. The inner `match` is, indeed, for list options. The intention was that only the lines beginning `| None...` and `| Some ...` be part of that match, but the next line has been caught up in it as well.

One simple solution is to use parentheses to make explicit the intended structure of the code.

```
# let rec zip_opt (xs : 'a list)
#                 (ys : 'b list)
#               : ('a * 'b) list option =
#   match xs, ys with
#   | [], [] -> Some []
#   | xhd :: xtl, yhd :: ytl ->
#       (match zip_opt xtl ytl with
#        | None -> None
#        | Some ztl -> Some ((xhd, yhd) :: ztl))
#   | _, _ -> None ;;
val zip_opt : 'a list -> 'b list -> ('a * 'b) list option = <fun>
```

Better yet is to make explicit the patterns that fall under the wildcard allowing them to move up in the ordering.

```
# let rec zip_opt (xs : 'a list)
#                 (ys : 'b list)
#               : ('a * 'b) list option =
#   match xs, ys with
#   | [], [] -> Some []
#   | [], _
#   | _, [] -> None
#   | xhd :: xtl, yhd :: ytl ->
#       match zip_opt xtl ytl with
#       | None -> None
#       | Some ztl -> Some ((xhd, yhd) :: ztl) ;;
val zip_opt : 'a list -> 'b list -> ('a * 'b) list option = <fun>
```

Exercise 79

Why is it necessary to make the patterns explicit before moving them up in the ordering? What goes wrong if we leave the pattern as `_, _`?

As an alternative, we can implement `zip` to raise an exception on lists of unequal length. Doing so simplifies the matches, since there's no issue of option poisoning.

```
# let rec zip (xs : 'a list)
#             (ys : 'b list)
#           : ('a * 'b) list =
#   match xs, ys with
#   | [], [] -> []
#   | [], _
#   | _, [] -> raise (Invalid_argument
#                       "zip: unequal length lists")
#   | xhd :: xtl, yhd :: ytl ->
#       (xhd, yhd) :: (zip xtl ytl) ;;
val zip : 'a list -> 'b list -> ('a * 'b) list = <fun>
```

Exercise 80

Define a function `zip_safe` that returns the zip of two equal-length lists, returning the empty list if the arguments are of unequal length. The implementation should call `zip`.

```
# zip_safe [1; 2; 3] [3; 2; 1] ;;
- : (int * int) list = [(1, 3); (2, 2); (3, 1)]
# zip_safe [1; 2; 3] [3; 2] ;;
- : (int * int) list = []
```

What problems do you see in this function?

10.3.3 Declaring new exceptions

Exceptions are first-class values, of the type `exn`. Like lists and options, exceptions have multiple value constructors. We've seen some already: `Exit`, `Failure`, `Invalid_argument`. (It's for that reason that we can pattern match against them in the `try...with` construct.)

Exceptions are exceptional in that new value constructors can be added dynamically. Here we define a new exception value constructor:

```
# exception Timeout ;;
exception Timeout
```

It turns out that this exception will be used in Chapter 17.

Exception constructors can take arguments. We define an `UnboundVariable` constructor that takes a string argument, used in Chapter 13, as

```
# exception UnboundVariable of string ;;
exception UnboundVariable of string
```

Exercise 81

In Section 6.6, we noted a problem with the definition of `fact` for computing the factorial function; it fails on negative inputs. Modify the definition of `fact` to raise an exception to make that limitation explicit.

Exercise 82

What are the types of the following expressions (or the values they define)?

1. `Some 42`
2. `[Some 42; None]`
3. `[None]`
4. `Exit`
5. `Failure "nth"`
6. `raise (Failure "nth")`
7. `raise`
8. `fun _ -> raise Exit`

9.
   ```
   let failwith s =
     raise (Failure s)
   ```

10.
    ```
    let sample x =
      failwith "not implemented"
    ```

11.
    ```
    let sample (x : int) (b : bool) : int list option =
      failwith "not implemented"
    ```

Problem 83

As in Problem 64, for each of the following OCaml function types define a function `f` (with no explicit typing annotations, that is, no uses of the `:` operator) for which OCaml would infer that type. (The functions need not be practical or do anything useful; they need only have the requested type.)

1. `int -> int -> int option`
2. `(int -> int) -> int option`
3. `'a -> ('a -> 'b) -> 'b`
4. `'a option list -> 'b option list -> ('a * 'b) list`

Problem 84

As in Problem 66, for each of the following function definitions of a function `f`, give a typing for the function that provides its most general type (as would be inferred by OCaml) or explain briefly why no type exists for the function.

1.
```
let rec f x =
  match x with
  | [] -> f
  | h :: t -> raise Exit ;;
```

2.
```
let f x =
  if x then (x, true)
  else (true, not x) ;;
```

Problem 85

Provide a more succinct definition of the function `f` from Problem 84(2), with the same type and behavior.

10.4 *Options or exceptions?*

Which should you use when writing code to handle anomalous conditions? Options or exceptions? This is a design decision. There is no universal right answer.

Anomalous conditions when running code cover a range of cases. One class of anomalies are conditions that should *never* occur, following from true bugs in code. For instance, when a function is applied to a set of arguments for which it was explicitly not defined – for example, applying the `median` function to an empty list, where the implementer of the median function has specified that it is not defined in that case – this constitutes a bug. The programmer who used the `median` function in that way has made a mistake. Unfortunately, the bug appears only at run time, when it is "too late". The best we can do in such cases is to abort the computation, returning control to some higher level for which recovery from the bug is possible (if such a higher level even exists), and providing as much information about the bug as possible. Some programming languages provide specific tools for such cases. In OCaml, exceptions are the right tool, raising an informative exception and hoping that a higher level can recover. And proper programming practice indicates doing just that.

For cases that are not simply bugs of this sort, that is, cases that are anomalous from the usual course yet expected to be handled, the choice between options and exceptions is governed by the properties of the two approaches.

Options are explicit: The type gives an indication that an anomaly might occur, and the compiler can make sure that such anomalies are handled. Exceptions are implicit: You (and the compiler) can't tell if an exception might be raised while executing a function. But exceptions are therefore more concise. The error handling doesn't impinge on the data and so doesn't poison every downstream use of the data. Code to handle the anomaly doesn't have to exist everywhere between where the problem occurs and where it's dealt with.

Which is more important, explicitness or concision? It depends.

- If the anomaly is a standard part of the computation, a frequent occurrence, that argues for making it explicit in an option type.
- If the anomaly is a rare occurrence, that argues for hiding it implicitly in the code.
- If the anomaly is localized to a small part of the code within which it can be handled, it makes sense to use an option type in that region.
- If the anomaly is ubiquitous, with the possibility of occurring anywhere in the code, the overhead of explicitly handling it everywhere in the code with an option type is likely too cumbersome. For example, a computation may run out of memory at more or less any point. It makes no sense to have a function return an option type, with `None` reserved for the case where the computation happened to run out of memory in the function. Rather, running out of memory is a natural use for an exception (and in fact, OCaml raises exceptions when it runs out of memory).

Is the anomalous occurrence a frequent case? Use options. A rare event? Use exceptions. Is the anomalous occurrence intrinsic to the conception? Use options. Extrinsic? Use exceptions.

Design decisions like this are ubiquitous. They are the bread and butter of the programming process. The precursor to making these decisions is possessing the tools that allow the alternative designs, the understanding of what the ramifications are, and the judgement to make a reasonable choice. The importance of having the choice is why, for instance, the `List` module provides both `nth` and `nth_opt`.

10.5 Unit testing with exceptions

In Section 6.7, we called for unit testing of functions to verify their correctness on representative inputs. Using the methodology of that section, we might write a unit testing function for `nth`, call it `nth_test`:

```
# let nth_test () =
#   unit_test (nth [5] 0 = 5) "nth singleton";
#   unit_test (nth [1; 2; 3] 0 = 1) "nth start";
#   unit_test (nth [1; 2; 3] 1 = 2) "nth middle" ;;
val nth_test : unit -> unit = <fun>
```

We run the tests by calling the function:

```
# nth_test () ;;
nth singleton passed
nth start passed
nth middle passed
- : unit = ()
```

The test function provides a report of the performance on all of the tests, showing that all tests are passed.

As mentioned in Section 6.7, we'll want to unit test `nth` as completely as is practicable, trying examples representing as wide a range of cases as possible. For instance, we might be interested in whether `nth` works in selecting the first, a middle, and the last element of a list. We've checked the first two of these conditions, but not the third. We can adjust the testing function accordingly:

```
# let nth_test () =
#   unit_test (nth [5] 0 = 5) "nth singleton";
#   unit_test (nth [1; 2; 3] 0 = 1) "nth start";
#   unit_test (nth [1; 2; 3] 1 = 2) "nth middle";
#   unit_test (nth [1; 2; 3] 2 = 3) "nth last" ;;
val nth_test : unit -> unit = <fun>
```

What about selecting at an index that is too large, as in the example `nth [1; 2; 3] 4`? We should make sure that `nth` works properly in this case as well. But what does "works properly" mean? According to the specification in the `List` module, `nth` should raise a `Failure` exception in this case. So we'll need a boolean expression that is `true` just in case evaluating the expression `nth [1; 2; 3] 4` raises the proper exception. We can achieve this by using a `try` ⟨⟩ `with` ⟨⟩ to trap any exception raised and verifying that it is the correct one. We might start with

```
# try nth [1; 2; 3] 4
# with
# | Failure _ -> true
# | _ -> false ;;
Line 3, characters 15-19:
3 | | Failure _ -> true
                   ^^^^
Error: This expression has type bool but an expression was expected
  of type
         int
```

but this fails to type-check, since the type of the `nth` expression is `int` (since it was applied to an `int list`), whereas the `with` clauses return a `bool`. We'll need to return a `bool` in the `try` as well. In fact, we should return `false`; if `nth [1; 2; 3] 4` manages to return a value and not raise an exception, that's a sign that `nth` has a bug! We revise the test condition to be

```
# try let _ = nth [1; 2; 3] 4 in
#     false
# with
# | Failure _ -> true
# | _ -> false ;;
- : bool = true
```

Adding this unit test to the unit testing function gives us

```
# let nth_test () =
#   unit_test (nth [5] 0 = 5) "nth singleton";
#   unit_test (nth [1; 2; 3] 0 = 1) "nth start";
#   unit_test (nth [1; 2; 3] 1 = 2) "nth middle";
#   unit_test (nth [1; 2; 3] 2 = 3) "nth last";
#   unit_test (try let _ = nth [1; 2; 3] 4 in
#                  false
#              with
#              | Failure _ -> true
#              | _ -> false) "nth index too big";;
val nth_test : unit -> unit = <fun>

# nth_test () ;;
nth singleton passed
nth start passed
nth middle passed
nth last passed
nth index too big passed
- : unit = ()
```

We'll later see more elegant ways to put together unit tests (Section 17.6).

Exercise 86

Augment `nth_test` to verify that `nth` works properly under additional conditions: on the empty list, with negative indexes, with lists other than integer lists, and so forth.

❧

With options and exceptions and their corresponding types, we've completed the introduction of the major compound data types that are built into the OCaml language. Table 10.1 provides a full list of these compound types, with their type constructors and value constructors. The advantages of compound types shouldn't be limited to built-ins though. In the next chapter, we'll extend the type system to allow user-defined compound types.

10.6 Supplementary material

- Lab 4: Error handling, options, and exceptions

Table 10.1: Built-in compound data types.

Type	*Type constructor*	*Value constructors*
functions	`⟨⟩ -> ⟨⟩`	`fun ⟨⟩ -> ⟨⟩`
tuples	`⟨⟩ * ⟨⟩`	`⟨⟩ , ⟨⟩`
	`⟨⟩ * ⟨⟩ * ⟨⟩`	`⟨⟩ , ⟨⟩ , ⟨⟩`
		`...`
lists	`⟨⟩ list`	`[]`
		`⟨⟩ :: ⟨⟩`
		`[ ⟨⟩ ; ⟨⟩ ; ...]`
records	`{ ⟨⟩ : ⟨⟩ ; ⟨⟩ : ⟨⟩ ; ...}`	`{ ⟨⟩ = ⟨⟩ ; ⟨⟩ = ⟨⟩ ; ...}`
options	`⟨⟩ option`	`None`
		`Some ⟨⟩`
exceptions	`exn`	`Exit`
		`Failure ⟨⟩`
		`...`
user-defined	*See Chapter 11*	