18 Testing

The goal of this chapter is to teach you how to write effective tests in OCaml, and to show off some tools that can help. Tooling is especially important in the context of testing because one of the things that prevents people from doing as much testing as they should is the tedium of it. But with the right tools in hand, writing tests can be lightweight and fun.

And that's important, because when testing is fun, you'll do more of it, and thorough testing is an essential element of building reliable and evolvable software. People sometimes imagine that tests are less important in a language like OCaml with a rich and expressive type-system, but in some sense the opposite is true. Really, types help you get more value out of your testing effort, both because they prevent you from needing to test all sorts of trivial properties that are automatically enforced by the type system, and because the rigidity of types mean that your code often has a kind of snap-together quality, where a relatively small number of tests can do an outsize amount to ensure that your code is behaving as expected.

Before we start introducing testing tools, it's worth pausing to consider what we want out of our tests in the first place.

Ideally, tests should be:

- **Easy to write and run**. Tests should require a minimum of boilerplate to create and to hook into your development process. Ideally, you should set things up so that tests are run automatically on every proposed change, preventing people from accidentally breaking the build.
- **Easy to update**. Tests that are hard to adjust in the face of code changes can become their own form of technical debt.
- **Fast**, so they don't slow down your development process.
- **Deterministic**. It's hard to take test failures seriously if there's a decent chance that the failure is a random glitch. You want your test failures to be believable indications of a problem, which requires determinism.
- **Understandable**. Good tests should be easy to read, and their failures should be localized and specific, so it's easy to find and fix the problem flagged by a failing test.

No testing framework can ensure that your tests satisfy these properties. But the tools you choose can help or hinder on all these fronts.

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

As we go through the rest of this chapter and introduce you to some of the available tooling, you should be able to see how each tool helps advance these goals.

18.1 Inline Tests

The first step towards a good testing environment is making it easy to set up and run a test. To that end, we'll show you how to write tests with `ppx_inline_test`, which lets you add tests to any module in a library with a specially annotated `let` binding.

To use inline tests in a library, we need to do two things:

- Tell Dune to expect inline tests to show up in the library, and
- enable `ppx_inline_test` as a preprocessor.

The first of these is achieved by adding an `(inline_tests)` declaration, and the second is achieved by adding `ppx_inline_test` to the set of preprocessors. Here's the resulting `dune` file.

```
(library
 (name foo)
 (libraries base stdio)
 (inline_tests)
 (preprocess (pps ppx_inline_test)))
```

With this done, any module in this library can host a test. We'll demonstrate this by creating a file called `test.ml`, containing one test.

```
open Base

let%test "rev" =
  List.equal Int.equal (List.rev [ 3; 2; 1 ]) [ 1; 2; 3 ]
```

The test passes if the expression on the right-hand side of the equals-sign evaluates to true. Inline tests are not automatically run with the instantiation of the module, but are instead registered for running via the test runner.

```
$ dune runtest
```

No output is generated because the test passed successfully. But if we break the test,

```
open Base

let%test "rev" =
  List.equal Int.equal (List.rev [ 3; 2; 1 ]) [ 3; 2; 1 ]
```

we'll see an error when we run it.

```
$ dune runtest
File "test.ml", line 3, characters 0-74: rev is false.

FAILED 1 / 1 tests
[1]
```

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

18.1.1 More Readable Errors with `test_eq`

One problem with the test output we just saw is that it doesn't show the data associated with the failed test, thus making it harder to diagnose and fix the problem when it occurs. We can fix this if we signal a test failure by throwing an exception, rather than by returning false. That exception can then be used to report the details of what went wrong.

To do this, we'll change our test declaration to use `let%test_unit` instead of `let%test`, so that the test no longer expects a body that returns a bool. We're also going to use the `[%test_eq]` syntax, which, given a type, generates code to test for equality and throw a meaningful exception if the arguments are unequal.

To use `[%test_eq]`, we're going to need to add the `ppx_assert` syntax extension, so we'll need to adjust our `dune` file appropriately.

```
(library
 (name foo)
 (libraries base stdio)
 (preprocess
  (pps ppx_inline_test ppx_assert))
 (inline_tests))
```

Here's what our new test looks like.

```
open Base

let%test_unit "rev" =
  [%test_eq: int list] (List.rev [ 3; 2; 1 ]) [ 3; 2; 1 ]
```

Now we can run the test to see what the output looks like.

```
$ dune runtest
File "test.ml", line 3, characters 0-79: rev threw
(duniverse/ppx_assert/runtime-lib/runtime.ml.E "comparison failed"
  ((1 2 3) vs (3 2 1) (Loc test.ml:4:13))).
  Raised at Base__Exn.protectx in file "duniverse/base/src/exn.ml",
    line 71, characters 4-114
  Called from
    Ppx_inline_test_lib__Runtime.time_and_reset_random_seeds in file
    "duniverse/ppx_inline_test/runtime-lib/runtime.ml", line 356,
    characters 15-52
  Called from Ppx_inline_test_lib__Runtime.test in file
    "duniverse/ppx_inline_test/runtime-lib/runtime.ml", line 444,
    characters 52-83

FAILED 1 / 1 tests
[1]
```

As you can see, the data that caused the comparison to fail is printed out, along with the stack backtrace. Sadly, the backtrace is in this case mostly a distraction. That's a downside of using exceptions to report test failures.

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

18.1.2 Where Should Tests Go?

The inline test framework lets you put tests into any `.ml` file that's part of a library. But just because you can do something doesn't mean you should.

Putting tests directly in the library you're building certainly has some benefits. For one thing, it lets you put a test for a given function directly after the definition of that function, which in some cases can be good for readability. This approach also lets you test aspects of your code that aren't exposed by its external interface.

While this sounds appealing at first glance, putting tests in libraries has several downsides.

- **Readability**. Including all of your tests directly in your application code can make that code itself harder to read. This can lead to people writing too few tests in an effort to keep their application code uncluttered.
- **Bloat**. When your tests are written as a part of your library, it means that every user of your library has to link in that testing code in their production application. Even though that code won't be run, it still adds to the size of the executable. It can also require dependencies on libraries that you don't need in production, which can reduce the portability of your code.
- **Testing mindset**. Writing tests on the inside of your libraries lets you write tests against any part of your implementation, rather than just the exposed API. This freedom is useful, but can also put you in the wrong testing mindset. Testing that's phrased in terms of the public API often does a better job of testing what's fundamental about your code, and will better survive refactoring of the implementation. Also, the discipline of keeping tests outside of requires you to write code that can be tested that way, which pushes towards better designs.

For all of these reasons, our recommendation is to put the bulk of your tests in test-only libraries created for that purpose. There are some legitimate reasons to want to put some test directly in your production library, e.g., when you need access to some functionality to do the test that's important but is really awkward to expose. But such cases are very much the exception.

Why Can't Inline Tests Go in Executables?

We've only talked about putting tests into libraries. What about executables? It turns out you can't do this directly, because Dune doesn't support the `inline_tests` declaration in source files that are directly part of an executable.

There's a good reason for this: the `ppx_inline_test` test runner needs to instantiate the modules that contain the tests. If those modules have toplevel side-effects, that's a recipe for disaster, since you don't want those top-level effects to be triggered by the test framework.

So, how do we test code that's part of an executable? The solution is to break up your program into two pieces: a directory containing a library that contains the logic of your program, but no top-level effects; and a directory for the executable that links in the library, and is responsible for launching the code.

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

18.2 Expect Tests

The tests we've shown so far have been mostly about checking some specific properties in a given scenario. Sometimes, though, what you want is not to test this or that property, but to capture and make visible your code's behavior. *Expect tests* let you do just that.

18.2.1 Basic Mechanics

With expect tests, your source file specifies both the code to be executed and the expected output. Upon running an expect test, any discrepancy between the expected output and what was actually generated is reported as a test failure.

Here's a simple example of a test written in this style. While the test generates output (though a call to `print_endline`), that output isn't captured in the source, at least, not yet.

```
open! Base
open Stdio

let%expect_test "trivial" = print_endline "Hello World!"
```

open and open!

In this example, we use `open!` instead of `open` because we happen not to be using any values from `Base`, and so the compiler will warn us about an unused open.

But because `Base` is effectively our standard library, we want to keep it open anyway, since we want any new code we write to find `Base`'s modules rather than those from the ordinary standard library. The exclamation point at the end of `open` suppresses that warning.

A sensible idiom is to always use `open!` when opening a library like `Base`, so that you don't have to choose when to use the `!`, and when not to.

If we run the test, we'll be presented with a diff between what we wrote, and a *corrected* version of the source file that now has an `[%expect]` clause containing the output. Note that Dune will use the `patdiff` tool if it's available, which generates easier-to-read diffs. You can install `patdiff` with `opam`.

```
$ dune runtest
     patdiff (internal) (exit 1)
(cd _build/default && rwo/_build/install/default/bin/patdiff
    -keep-whitespace -location-style omake -ascii test.ml
    test.ml.corrected)
------ test.ml
++++++ test.ml.corrected
File "test.ml", line 5, characters 0-1:
 |open! Base
 |open Stdio
 |
 |let%expect_test "trivial" =
-|  print_endline "Hello World!"
+|  print_endline "Hello World!";
```

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

```
+|  [%expect {| Hello World! |}]
[1]
```

The expect test runner also creates a version of the file with the captured output, with `.corrected` appended to the end of the filename. If this new output looks correct, we can *promote* it by copying the corrected file over the original source. The `dune promote` command does just this, leaving our source as follows.

```
open Base
open Stdio

let%expect_test "trivial" =
  print_endline "Hello World!";
  [%expect {| Hello World! |}]
```

Now, if we run the test again, we'll see that it passes.

```
$ dune runtest
```

We only have one expect block in this example, but the system supports having multiple expect blocks:

```
open Base
open Stdio

let%expect_test "multi-block" =
  print_endline "Hello";
  [%expect {| Hello |}];
  print_endline "World!";
  [%expect {| World! |}]
```

18.2.2 What Are Expect Tests Good For?

It's not obvious why one would want to use expect tests in the first place. Why should this:

```
open Base
open Stdio

let%expect_test _ =
  print_s [%sexp (List.rev [ 3; 2; 1 ] : int list)];
  [%expect {| (1 2 3) |}]
```

be preferable to this:

```
open Base

let%test "rev" =
  List.equal Int.equal (List.rev [ 3; 2; 1 ]) [ 1; 2; 3 ]
```

Indeed, for examples like this, expect tests aren't better: simple example-based tests like the one above work fine when it's easy and convenient to write out specific examples in full. And, as we'll discuss later in the chapter, *property tests* are your best bet when you have a clear set of predicates that you want to test, and examples can be naturally generated at random.

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

Where expect tests shine is where you want to make visible some aspect of the behavior of your system that's hard to capture in a predicate. This is more useful than it might seem at first. Let's consider a few different example use-cases to see why.

18.2.3 Exploratory Programming

Expect tests can be especially helpful when you're in exploration mode, where you're trying to solve a problem by playing around with the data, and have no clear specification in advance.

A common programming task of this type is web-scraping, where the goal is generally to extract some useful information from a web page. Figuring out the right way to do so often involves trial and error.

Here's some code that does this kind of data extraction, using the `lambdasoup` package to traverse a chunk of HTML and spit out some data embedded within it. In particular, the function aims to produce the set of hosts that show up in links within a document.

```
open Base
open Stdio

let get_href_hosts soup =
  Soup.select "a[href]" soup
  |> Soup.to_list
  |> List.map ~f:(Soup.R.attribute "href")
  |> Set.of_list (module String)
```

We can use an expect test to demonstrate what this function does on an example page.

```
let%expect_test _ =
  let example_html =
    {|
    <html>
      Some random <b>text</b> with a
      <a href="http://ocaml.org/base">link</a>.
      And here's another
      <a href="http://github.com/ocaml/dune">link</a>.
      And here is <a>link</a> with no href.
    </html>|}
  in
  let soup = Soup.parse example_html in
  let hrefs = get_href_hosts soup in
  print_s [%sexp (hrefs : Set.M(String).t)]
```

Quoted Strings

The example above used a new syntax for string literals, called *quoted strings*. Here's an example.

```
# {|This is a quoted string|};;
- : string = "This is a quoted string"
```

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

The advantage of this syntax is that it allows the content to be written without the usual escaping required for ordinary string literals. Consider the following examples.

```
# {|This is a literal quote: "|};;
((:- : string = "This is a literal quote: \""::)
```

As you can see, we didn't need to escape the included quote, though the version of the string echoed back by the toplevel uses ordinary string literal syntax, and so the quote there comes out escaped.

Quoted strings are especially useful when writing strings containing text from another language, like HTML. With quoted strings, you can just paste in a snippet of some other source language, and it should work unmodified.

The one tricky corner is if you need to include a literal `|}` inside your quoted string. The trick is that you can change the delimiter for the quoted string by adding an arbitrary identifier, thereby ensuring that the delimiter won't show up in the body of the string.

```
# {xxx|This is how you quote a {|quoted string|}|xxx};;
- : string = "This is how you quote a {|quoted string|}"
```

If we run the test, we'll see that the output isn't exactly what was intended.

```
$ dune runtest
     patdiff (internal) (exit 1)
...
------ test.ml
++++++ test.ml.corrected
File "test.ml", line 24, characters 0-1:
 |  |> List.map ~f:(Soup.R.attribute "href")
 |  |> Set.of_list (module String)
 |
 |[@@@part "1"] ;;
 |let%expect_test _ =
 |  let example_html = {|
 |    <html>
 |      Some random <b>text</b> with a
 |      <a href="http://ocaml.org/base">link</a>.
 |      And here's another
 |      <a href="http://github.com/ocaml/dune">link</a>.
 |      And here is <a>link</a> with no href.
 |    </html>|}
 |  in
 |  let soup = Soup.parse example_html in
 |  let hrefs = get_href_hosts soup in
-|  print_s [%sexp (hrefs : Set.M(String).t)]
+|  print_s [%sexp (hrefs : Set.M(String).t)];
+|  [%expect {| (http://github.com/ocaml/dune http://ocaml.org/base)
     |}]
[1]
```

The problem here is that we failed to extract the host from the URI string. I.e., we ended up with `http://github.com/ocaml/dune` instead of `github.com`. We can fix that by using the `uri` library to parse the string and extract the host. Here's the modified code.

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

```
let get_href_hosts soup =
  Soup.select "a[href]" soup
  |> Soup.to_list
  |> List.map ~f:(Soup.R.attribute "href")
  |> List.filter_map ~f:(fun uri -> Uri.host (Uri.of_string uri))
  |> Set.of_list (module String)
```

And if we run the test again, we'll see that the output is now as it should be.

```
$ dune runtest
     patdiff (internal) (exit 1)
...
------ test.ml
++++++ test.ml.corrected
File "test.ml", line 26, characters 0-1:
 |  |> Set.of_list (module String)
 |
 |[@@@part "1"] ;;
 |let%expect_test _ =
 |  let example_html = {|
 |    <html>
 |      Some random <b>text</b> with a
 |      <a href="http://ocaml.org/base">link</a>.
 |      And here's another
 |      <a href="http://github.com/ocaml/dune">link</a>.
 |      And here is <a>link</a> with no href.
 |    </html>|}
 |  in
 |  let soup = Soup.parse example_html in
 |  let hrefs = get_href_hosts soup in
 |  print_s [%sexp (hrefs : Set.M(String).t)];
-|  [%expect {| (http://github.com/ocaml/dune http://ocaml.org/base)
    |}]
+|  [%expect {| (github.com ocaml.org) |}]
[1]
```

One nice aspect of this exploratory workflow is that once you've gotten things working, you can leave the examples you used to develop the code as permanent tests.

18.2.4 Visualizing Complex Behavior

Expect tests can be used to examine the dynamic behavior of a system. Let's walk through a simple example: a rate limiter. The job of a rate limiter is to bound the rate at which a system consumes a particular resource. The following is the `mli` for a library that specifies the logic of a simple rolling-window-style rate limiter, where the intent is to make sure that there's no window of time of the specified period during which more than a specified number of events occurs.

```
open Core

type t

val create : now:Time_ns.t -> period:Time_ns.Span.t -> rate:int -> t
val maybe_consume : t -> now:Time_ns.t -> [ `Consumed | `No_capacity ]
```

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

We can demonstrate the behavior of the system by running through some examples. First, we'll write some helper functions to make the examples shorter and easier to read.

```
open Core

let start_time = Time_ns.of_string "2021-06-01 7:00:00"

let limiter () =
  Rate_limiter.create
    ~now:start_time
    ~period:(Time_ns.Span.of_sec 1.)
    ~rate:2

let consume lim offset =
  let result =
    Rate_limiter.maybe_consume
      lim
      ~now:(Time_ns.add start_time (Time_ns.Span.of_sec offset))
  in
  printf
    "%4.2f: %s\n"
    offset
    (match result with
    | `Consumed -> "C"
    | `No_capacity -> "N")
```

Here, we define three values: `start_time`, which is just a point in time at which to begin our examples; `limiter`, which is a function for constructing a fresh `Limiter.t` object, with some reasonable defaults; and `consume`, which attempts to consume a resource.

Notably, `consume` doesn't just update the limiter, it also prints out a marker of the result, i.e., whether the consumption succeeded or failed.

Now we can use these helpers to see how the rate limiter would behave in a simple scenario. First, we're going to try to consume three times at time zero; then we're going to wait a half-second and consume again, and then we'll wait one more half-second, and try again.

```
let%expect_test _ =
  let lim = limiter () in
  let consume offset = consume lim offset in
  (* Exhaust the rate limit, without advancing the clock. *)
  for _ = 1 to 3 do
    consume 0.
  done;
  [%expect {| |}];
  (* Wait until a half-second has elapsed, try again *)
  consume 0.5;
  [%expect {| |}];
  (* Wait until a full second has elapsed, try again *)
  consume 1.;
  [%expect {|  |}]
```

Running the tests and accepting the promotions will include the execution trace.

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

```
let%expect_test _ =
  let lim = limiter () in
  let consume offset = consume lim offset in
  (* Exhaust the rate limit, without advancing the clock. *)
  for _ = 1 to 3 do
    consume 0.
  done;
  [%expect {|
    0.00: C
    0.00: C
    0.00: C |}];
  (* Wait until a half-second has elapsed, try again *)
  consume 0.5;
  [%expect {| 0.50: C |}];
  (* Wait until a full second has elapsed, try again *)
  consume 1.;
  [%expect {| 1.00: C |}]
```

The above, however, is not the expected outcome! In particular, all of our calls to `consume` succeeded, despite us violating the 2-per-second rate limit. That's because there was a bug in our implementation. The implementation has a queue of times where consume events occurred, and we use this function to drain the queue.

```
let rec drain_old_events t =
  match Queue.peek t.events with
  | None -> ()
  | Some time ->
    if Time_ns.Span.( < ) (Time_ns.diff t.now time) t.period
    then (
      ignore (Queue.dequeue_exn t.events : Time_ns.t);
      drain_old_events t)
```

But the comparison goes the wrong way: we should discard events that are older than the limit-period, not younger. If we fix that, we'll see that the trace behaves as we'd expect.

```
let%expect_test _ =
  let lim = limiter () in
  let consume offset = consume lim offset in
  (* Exhaust the rate limit, without advancing the clock. *)
  for _ = 1 to 3 do
    consume 0.
  done;
  [%expect {|
    0.00: C
    0.00: C
    0.00: N |}];
  (* Wait until a half-second has elapsed, try again *)
  consume 0.5;
  [%expect {| 0.50: N |}];
  (* Wait until a full second has elapsed, try again *)
  consume 1.;
  [%expect {| 1.00: C |}]
```

One of the things that makes this test readable is that we went to some trouble to keep the code short and easy to read. Some of this was about creating useful helper

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

functions, and some of it was about having a concise and noise-free format for the data captured by the expect blocks.

18.2.5 End-to-End Tests

The expect tests we've seen so far have been self-contained, not doing any IO or interacting with system resources. As a result, these tests are fast to run and entirely deterministic.

That's a great ideal, but it's not always achievable, especially when you want to run more end-to-end tests of your program. But even if you need to run tests that involve multiple processes interacting with each other and using real IO, expect tests are still a useful tool.

To see how such tests can be built, we'll write some tests for the echo server we developed in Chapter 17.2 (Example: An Echo Server).

We'll start by creating a new test directory with a dune file next to our echo-server implementation.

```
(library
 (name echo_test)
 (libraries core async)
 (preprocess (pps ppx_jane))
 (inline_tests (deps ../bin/echo.exe)))
```

The important line is the last one, where in the `inline_tests` declaration, we declare a dependency on the echo-server binary. Also, note that rather than select useful preprocessors one by one, we used the omnibus `ppx_jane` package, which bundles together a collection of useful extensions.

That done, our next step is to write some helper functions. We won't show the implementation, but here's the signature for our `Helpers` module. Note that there's an argument in the `launch` function that lets you enable the feature in the echo server that causes it to uppercase the text it receives.

```
open! Core
open Async

(** Launches the echo server *)
val launch : port:int -> uppercase:bool -> Process.t Deferred.t

(** Connects to the echo server, returning a reader and writer for
   communicating with the server. *)
val connect : port:int -> (Reader.t * Writer.t) Deferred.t

(** Sends data to the server, printing out the result  *)
val send_data : Reader.t -> Writer.t -> string -> unit Deferred.t

(** Kills the echo server, and waits until it exits  *)
val cleanup : Process.t -> unit Deferred.t
```

With the above, we can now write a test that launches the server, connects to it over TCP, and then sends some data and displays the results.

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

```
open! Core
open Async
open Helpers

let%expect_test "test uppercase echo" =
  let port = 8081 in
  let%bind process = launch ~port ~uppercase:true in
  Monitor.protect
    (fun () ->
      let%bind r, w = connect ~port in
      let%bind () = send_data r w "one two three\n" in
      let%bind () = [%expect] in
      let%bind () = send_data r w "one 2 three\n" in
      let%bind () = [%expect] in
      return ())
    ~finally:(fun () -> cleanup process)
```

Note that we put in some expect annotations where we want to see data, but we haven't filled them in. We can now run the test to see what happens. The results, however, are not what you might hope for.

```
$ dune runtest
Entering directory
    'rwo/_build/default/book/testing/examples/erroneous/echo_test_original'
     patdiff (internal) (exit 1)
(cd _build/default && rwo/_build/install/default/bin/patdiff
    -keep-whitespace -location-style omake -ascii test/test.ml
    test/test.ml.corrected)
------ test/test.ml
++++++ test/test.ml.corrected
File "test/test.ml", line 11, characters 0-1:
 |open! Core
 |open Async
 |open Helpers
 |
 |let%expect_test "test uppercase echo" =
 |  let port = 8081 in
 |  let%bind process  = launch ~port ~uppercase:true in
 |  Monitor.protect (fun () ->
 |      let%bind (r,w) = connect ~port in
 |      let%bind () = send_data r w "one two three\n" in
-|      let%bind () = [%expect] in
+|      let%bind () = [%expect.unreachable] in
 |      let%bind () = send_data r w "one 2 three\n" in
-|      let%bind () = [%expect] in
+|      let%bind () = [%expect.unreachable] in
 |      return ())
 |    ~finally:(fun () -> cleanup process)
+|[@@expect.uncaught_exn {|
+|  (* CR expect_test_collector: This test expectation appears to
    contain a backtrace.
+|     This is strongly discouraged as backtraces are fragile.
+|     Please change this test to not include a backtrace. *)
+|
+|  (monitor.ml.Error
+|    (Unix.Unix_error "Connection refused" connect 127.0.0.1:8081)
```

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

```
+|    ("<backtrace elided in test>" "Caught by monitor
    Tcp.close_sock_on_error"))
+|  Raised at Base__Result.ok_exn in file
    "duniverse/base/src/result.ml", line 201, characters 17-26
+|  Called from Expect_test_collector.Make.Instance.exec in file
    "duniverse/ppx_expect/collector/expect_test_collector.ml", line
    244, characters 12-19 |}]
[1]
```

What went wrong here? The issue is that the connect fails, because at the time of the connection, the echo server hasn't finished setting up the server. We can fix this by adding a one second delay before connecting, using Async's `Clock.after`. With this change, the test now passes, with the expected results.

```
open! Core
open Async
open Helpers

let%expect_test "test uppercase echo" =
  let port = 8081 in
  let%bind process  = launch ~port ~uppercase:true in
  Monitor.protect (fun () ->
      let%bind () = Clock.after (Time.Span.of_sec 1.) in
      let%bind (r,w) = connect ~port in
      let%bind () = send_data r w "one two three\n" in
      let%bind () = [%expect{| ONE TWO THREE |}] in
      let%bind () = send_data r w "one 2 three\n" in
      let%bind () = [%expect{| ONE 2 THREE |}] in
      return ())
    ~finally:(fun () -> cleanup process)
```

We fixed the problem, but the solution should make you uncomfortable. For one thing, why is one second the right timeout, rather than a half a second, or ten? The time we wait is some balance between reducing the likelihood of a non-deterministic failure versus preserving performance of the test, which is a bit of an awkward trade-off to have to make.

We can improve on this by removing the `Clock.after` call, and instead adding a retry loop to the `connect` test helper

```
let rec connect ~port =
  match%bind
    Monitor.try_with (fun () ->
        Tcp.connect
          (Tcp.Where_to_connect.of_host_and_port
             { host = "localhost"; port }))
  with
  | Ok (_, r, w) -> return (r, w)
  | Error _ ->
    let%bind () = Clock.after (Time.Span.of_sec 0.01) in
    connect ~port
```

There's still a timeout in this code, in that we wait a bit before retrying. But that timeout is quite aggressive, so you never waste more than 10 milliseconds waiting unnecessarily. That means tests will typically run fast, but if they do run slowly (maybe because your machine is heavily loaded during a big build), the test will still pass.

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

The lesson here is that keeping tests deterministic in the context of running programs doing real I/O gets messy fast. When possible, you should write your code in a way that allows most of it to be tested without having to connect to real running servers. But when you do need to do it, you can still use expect tests for this purpose.

18.2.6 How to Make a Good Expect Test

Taken together, these examples suggest some guidelines for building good expect tests:

- **Write helper functions** to help you set up your test scenarios more concisely.
- **Write custom pretty-printers** that surface just the information that you need to see in the test. This makes your tests easier to read, and also minimizes unnecessary churn when details that are irrelevant to your test change.
- **Aim for determinism**, ideally by organizing your code so you can put it through its paces without directly interacting with the outside world, which is generally the source of non-determinism. But if you must, be careful to avoid timeouts and other stopgaps that will fall apart under performance pressure.

18.3 Property Testing with Quickcheck

Many tests amount to little more than individual examples decorated with simple assertions to check this or that property. *Property testing* is a useful extension of this approach, which lets you explore a much larger portion of your code's behavior with only a small amount of programmer effort.

The basic idea is simple enough. A property test requires two things: a function that takes an example input and checks that a given property holds on that example; and a way of generating random examples. The test then checks whether the predicate holds over many randomly generated examples.

We can write a property test using only the tools we've learned so far. In this example, we'll check an obvious-seeming invariant connecting three operations:

- `Int.sign`, which computes a `Sign.t` representing the sign of an integer, either `Positive`, `Negative`, or `Zero`
- `Int.neg`, which negates a number
- `Sign.flip`, which flips a `Sign.t`, i.e., mapping `Positive` to `Negative` and vice versa.

The invariant we want to check is that the sign of the negation of any integer `x` is the flip of the sign of `x`.

Here's a simple implementation of this test.

```
open Base

let%test_unit "negation flips the sign" =
  for _ = 0 to 100_000 do
    let x = Random.int_incl Int.min_value Int.max_value in
    [%test_eq: Sign.t]
```

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

```
      (Int.sign (Int.neg x))
      (Sign.flip (Int.sign x))
  done
```

As you might expect, the test passes.

```
$ dune runtest
```

One choice we had to make in our implementation is which probability distribution to use for selecting examples. This may not seem like an important question, but it is. When it comes to testing, not all probability distributions are created equal.

Indeed, the choice we made, which was to pick integers uniformly and at random from the full set of integers, is problematic, since it picks interesting special cases, like zero and one, with the same probability as everything else. Given the number of integers, the chance of testing any of those special cases is rather low, which seems like a problem.

This is a place where `Quickcheck` can help. `Quickcheck` is a library to help automate the construction of testing distributions. Let's try rewriting the above example using it. Note that we open `Core` here because `Core` has nicely integrated support for `Quickcheck`, with helper functions already integrated into most common modules. There's also a standalone `Base_quickcheck` library that can be used without `Core`.

```
open Core

let%test_unit "negation flips the sign" =
  Quickcheck.test
    ~sexp_of:[%sexp_of: int]
    (Int.gen_incl Int.min_value Int.max_value)
    ~f:(fun x ->
      [%test_eq: Sign.t]
        (Int.sign (Int.neg x))
        (Sign.flip (Int.sign x)))
```

Note that we didn't explicitly state how many examples should be tested. Quickcheck has a built-in default which can be overridden by way of an optional argument.

Running the test uncovers the fact that the property we've been testing doesn't actually hold on all outputs, as you can see below.

```
$ dune runtest
File "test.ml", line 3, characters 0-244: negation flips the sign
    threw
("Base_quickcheck.Test.run: test failed" (input -4611686018427387904)
  (error
    ((duniverse/ppx_assert/runtime-lib/runtime.ml.E "comparison
    failed"
       (Neg vs Pos (Loc test.ml:7:19)))
      "Raised at Ppx_assert_lib__Runtime.failwith in file
    \"duniverse/ppx_assert/runtime-lib/runtime.ml\", line 28,
    characters 28-53\
      \nCalled from Base__Or_error.try_with in file
    \"duniverse/base/src/or_error.ml\", line 76, characters 9-15\
      \n"))).
  Raised at Base__Exn.protectx in file "duniverse/base/src/exn.ml",
    line 71, characters 4-114
```

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

```
  Called from
    Ppx_inline_test_lib__Runtime.time_and_reset_random_seeds in file
    "duniverse/ppx_inline_test/runtime-lib/runtime.ml", line 356,
    characters 15-52
  Called from Ppx_inline_test_lib__Runtime.test in file
    "duniverse/ppx_inline_test/runtime-lib/runtime.ml", line 444,
    characters 52-83

FAILED 1 / 1 tests
[1]
```

The example that triggers the exception is `-4611686018427387904`, also known as `Int.min_value`, which is the smallest value of type `Int.t`. This uncovers something about integers which may not have been obvious, which is that the largest int, `Int.max_value`, is smaller in absolute value than `Int.max_value`.

```
# Int.min_value;;
- : int = -4611686018427387904
# Int.max_value;;
- : int = 4611686018427387903
```

That means there's no natural choice for the negation of `min_value`. It turns out that the standard behavior here (not just for OCaml) is for the negation of `min_value` to be equal to itself.

```
# Int.neg Int.min_value;;
- : int = -4611686018427387904
```

Quickcheck's decision to put much larger weight on special cases is what allowed us to discover this unexpected behavior. Note that in this case, it's not really a bug that we've uncovered, it's just that the property that we thought would hold can't in practice. But either way, Quickcheck helped us understand the behavior of our code better.

18.3.1 Handling Complex Types

Tests can't subsist on simple atomic types alone, which is why you'll often want to build probability distributions over more complex types. Here's a simple example, where we want to test the behavior of `List.rev_append`. For this test, we're going to use a probability distribution for generating pairs of lists of integers. The following example shows how that can be done using Quickcheck's combinators.

```
open Core

let gen_int_list_pair =
  let int_list_gen =
    List.gen_non_empty (Int.gen_incl Int.min_value Int.max_value)
  in
  Quickcheck.Generator.both int_list_gen int_list_gen

let%test_unit "List.rev_append is List.append of List.rev" =
  Quickcheck.test
    ~sexp_of:[%sexp_of: int list * int list]
    gen_int_list_pair
```

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

```
    ~f:(fun (l1, l2) ->
      [%test_eq: int list]
        (List.rev_append l1 l2)
        (List.append (List.rev l1) l2))
```

Here, we made use of `Quickcheck.Generator.both`, which is useful for creating a generator for pairs from two generators for the constituent types.

```
# open Core;;
# #show Quickcheck.Generator.both;;
val both :
  'a Base_quickcheck.Generator.t ->
  'b Base_quickcheck.Generator.t -> ('a * 'b)
    Base_quickcheck.Generator.t
```

The declaration of the generator is pretty simple, but it's also tedious. Happily, Quickcheck ships with a PPX that can automate creation of the generator given just the type declaration. We can use that to simplify our code, as shown below.

```
open Core

let%test_unit "List.rev_append is List.append of List.rev" =
  Quickcheck.test
    ~sexp_of:[%sexp_of: int list * int list]
    [%quickcheck.generator: int list * int list]
    ~f:(fun (l1, l2) ->
      [%test_eq: int list]
        (List.rev_append l1 l2)
        (List.append (List.rev l1) l2))
```

This also works with other, more complex data-types, like variants. Here's a simple example.

```
type shape =
  | Circle of { radius: float }
  | Rect of { height: float; width: float }
  | Poly of (float * float) list
[@@deriving quickcheck];;
```

This will make a bunch of reasonable default decisions, like picking `Circle`, `Rect`, and `Poly` with equal probability. We can use annotations to adjust this, for example, by specifying the weight on a particular variant.

```
type shape =
  | Circle of { radius: float } [@quickcheck.weight 0.5]
  | Rect of { height: float; width: float }
  | Poly of (float * float) list
[@@deriving quickcheck];;
```

Note that the default weight on each case is `1`, so now `Circle` will be generated with probability `0.5 / 2.5` or `0.2`, instead of the 1/3rd probability that it would have natively.

18.3.2 More Control with Let-Syntax

If the annotations associated with `ppx_quickcheck` don't let you do precisely what you want, you can get more control by taking advantage of the fact that Quickcheck's

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

generators form a monad. That means it supports operators like `bind` and `map`, which we first presented in an error handling context in Chapter 8.1.3 (`bind` and Other Error Handling Idioms).

In combination with let syntax, the generator monad gives us a convenient way to specify generators for custom types. Here's an example generator for the `shape` type above.

```
# let gen_shape =
    let open Quickcheck.Generator.Let_syntax in
    let module G = Base_quickcheck.Generator in
    let circle =
      let%map radius = G.float_positive_or_zero in
      Circle { radius }
    in
    let rect =
      let%bind height = G.float_positive_or_zero in
      let%map width = G.float_inclusive height Float.infinity in
      Rect { height; width }
    in
    let poly =
      let%map points =
        List.gen_non_empty
          (G.both G.float_positive_or_zero G.float_positive_or_zero)
      in
      Poly points
    in
    G.union [ circle; rect; poly ];;
val gen_shape : shape Base_quickcheck.Generator.t = <abstr>
```

Throughout this function we're making choices about the probability distribution. For example, the use of the `union` operator means that circles, rectangles and polygons will be equally likely. We could have used `weighted_union` to pick a different distribution. Also, we've ensured that all float values are non-negative, and that the width of the rectangle is no smaller than its height.

The full API for building generators is beyond the scope of this chapter, but it's worth digging into the API docs if you want more control over the distribution of your test examples.

18.4 Other Testing Tools

The testing tools we've described in this chapter cover a lot of ground, but there are other tools worth knowing about.

18.4.1 Other Tools to Do (Mostly) the Same Things

Here are some notable tools that do more or less the same things as the testing tools we've featured in this chapter.

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press

- Alcotest[1], which is another system for registering and running tests.
- qcheck[2], an alternative implementation of quickcheck.
- Dune's cram tests[3], which are expect-like tests that are written in a shell-like syntax. These are great for testing command-line utilities, and are inspired by Mercurial's testing framework.

Which of these you might end up preferring is to some degree a matter of taste.

18.4.2 Fuzzing

There's one other kind of testing tool that we haven't covered in this chapter, but is worth knowing about: *instrumentation-guided fuzzing*. You can think of this as another take on property testing, with a very different approach to generating random examples.

Traditional fuzzing just throws randomly mutated data at a program, and looks for some indication of failure, often simply the program crashing with a segfault. This kind of fuzzing has been surprisingly effective at finding bugs, especially security bugs, in production software. But blind randomization is still quite limited in terms of how much program behavior it can effectively explore.

Instrumentation-guided fuzzing improves on this by instrumenting the program, and then using that instrumentation to guide the randomization in the direction of more code coverage. By far the most successful tool in this space is American Fuzzy Lop[4], or AFL, and OCaml has support for the necessary instrumentation.

AFL can have eerily good results, and can with no guidance do things like constructing nearly-parseable text when fuzzing a parser, just by iteratively randomizing inputs in the direction of more coverage of the program being exercised.

If you're interested in AFL, there are some related tools worth knowing about.

- Crowbar[5] is a quickcheck-style library for writing down properties to be tested by AFL.
- Bun[6] is a library for integrating AFL into your continuous-integration pipeline.

[1] https://github.com/mirage/alcotest
[2] https://github.com/c-cube/qcheck
[3] https://dune.readthedocs.io/en/stable/tests.html#cram-tests
[4] https://github.com/google/AFL
[5] https://github.com/stedolan/crowbar
[6] https://github.com/ocurrent/bun

https://doi.org/10.1017/9781009129220.021 Published online by Cambridge University Press