

---

# Dune Documentation

**J r mie Dimino**

**Jul 10, 2026**



# CONTENTS

<b>1</b>	<b>Getting Started and Core Concepts</b>	<b>3</b>
1.1	Overview . . . . .	3
1.2	Quickstart . . . . .	5
1.3	Command-Line Interface . . . . .	12
<b>2</b>	<b>Tutorials</b>	<b>21</b>
2.1	Developing with Dune . . . . .	21
2.2	OCaml Package Management With Dune . . . . .	56
2.3	OxCaml Parameterised Libraries with Dune . . . . .	64
<b>3</b>	<b>How-to Guides</b>	<b>69</b>
3.1	How to Install Dune . . . . .	69
3.2	How to Set up Automatic Formatting . . . . .	69
3.3	How to Generate Opam Files from <code>dune-project</code> . . . . .	71
3.4	Cross-Compilation . . . . .	73
3.5	Dealing with Foreign Libraries . . . . .	74
3.6	Generating Documentation . . . . .	80
3.7	How to Load Additional Files at Runtime . . . . .	82
3.8	Instrumentation . . . . .	87
3.9	JavaScript Compilation With <code>Js_of_ocaml</code> . . . . .	89
3.10	Wasm Compilation With <code>Wasm_of_ocaml</code> . . . . .	90
3.11	JavaScript Compilation With Melange . . . . .	91
3.12	Virtual Libraries . . . . .	96
3.13	Writing and Running Tests . . . . .	98
3.14	How to Bundle Resources . . . . .	107
3.15	How to Handle External System Dependencies ( <code>depexts</code> ) . . . . .	108
3.16	How to Load a Project in a Toplevel . . . . .	109
3.17	Using Rule Generation . . . . .	110
3.18	How to Override the Default C Entrypoint With C Stubs . . . . .	114
3.19	How to Organize Multiple Executables in Different Directories . . . . .	115
3.20	How to Release a Binary Distribution of an Application on GitHub with Dune . . . . .	117
3.21	How to make a Homebrew Package with Dune . . . . .	118
3.22	How to Use Opam Alongside Dune Package Management . . . . .	120
3.23	How to Configure Editors with Package Management . . . . .	120
3.24	How to Set Up Shell Command Completion . . . . .	121
3.25	How to set up Continuous Integration with Package Management . . . . .	122
3.26	How to Customize Lock Directories of Developer Tools . . . . .	123
3.27	How to Use Multiple Build Contexts . . . . .	124
<b>4</b>	<b>Reference Manual</b>	<b>129</b>

4.1	dune	129
4.2	dune-project	161
4.3	dune-workspace	173
4.4	config	180
4.5	Lexical Conventions	183
4.6	Actions	185
4.7	Ordered Set Language	191
4.8	Boolean Language	191
4.9	Predicate Language	192
4.10	Library Dependencies	192
4.11	Preprocessing Specification	194
4.12	Cram Tests	196
4.13	Variables	199
4.14	Dependency Specification	204
4.15	OCaml Flags	206
4.16	Sandboxing	206
4.17	Locks	207
4.18	Diffing and Promotion	208
4.19	Diffing	208
4.20	Promotion	209
4.21	Package Specification	209
4.22	Aliases	211
4.23	Foreign Stubs	216
4.24	Foreign Archives	217
4.25	Extra Objects	218
4.26	Flags in Foreign Code	219
4.27	Command Line Interface	219
4.28	dune tools	223
4.29	Dune Libraries	225
4.30	Dune Caches	228
4.31	How Caching Works	228
4.32	Configuration	228
4.33	Cache Storage Mode	229
4.34	Trimming the Cache	229
4.35	Reproducibility	230
4.36	The Rocq Prover Build Language	230
4.37	Dune RPC	242
4.38	Packages	243
4.39	Findlib Integration	244
<b>5</b>	<b>Explanation</b>	<b>245</b>
5.1	The Dune Mental Model	245
5.2	Dune Projects and Workspaces	248
5.3	How Preprocessing Works	248
5.4	The OCaml Ecosystem	249
5.5	How Package Management Works	250
5.6	How Dune integrates with opam	254
5.7	How Dune Uses Dune to Build Dune	256
5.8	A Tour of the Dune Codebase	257
<b>6</b>	<b>Advanced Topics</b>	<b>263</b>
6.1	Dynamic Loading of Packages with Findlib	263
6.2	Profiling Dune	264
6.3	Package Version	264

6.4	OCaml Syntax . . . . .	264
6.5	Variables for Artifacts . . . . .	265
6.6	Building an Ad Hoc .cmxs . . . . .	265
<b>7</b>	<b>Miscellaneous</b>	<b>267</b>
7.1	FAQ . . . . .	267
7.2	Goal of Dune . . . . .	270
7.3	Working on the Dune Codebase . . . . .	272
	<b>Index</b>	<b>285</b>



Dune is a build system for OCaml projects. Using it, you can build executables, libraries, run tests, and much more.



## GETTING STARTED AND CORE CONCEPTS

These documents should be the first ones read by new Dune users. They explain what Dune is, how it works, and how to use it.

### 1.1 Overview

#### 1.1.1 Introduction

Dune is a build system for OCaml (with support for Reason and Rocq). It is not intended as a completely generic build system that's able to build any project in any language. On the contrary, it makes lots of choices in order to encourage a consistent development style.

This scheme is inspired from the one used inside Jane Street and adapted to the opam world. It has matured over a long time and is used daily by hundreds of developers, which means that it is highly tested and productive.

When using Dune, you give very little, high-level information to the build system, which in turn takes care of all the low-level details from the compilation of your libraries, executables, and documentation to the installation, setting up of tests, and setting up development tools such as Merlin, etc.

In addition to the normal features expected from an OCaml build system, Dune provides a few additional ones that separate it from the crowd:

- You never need to tell Dune the location of things such as libraries. Dune will discover them automatically. In particular, this means that when you want to reorganise your project, you need nothing other than to rename your directories, Dune will do the rest.
- Things always work the same whether your dependencies are local or installed on the system. In particular, this means that you can insert the source for a project dependency in your working copy, and Dune will start using it immediately. This makes Dune a great choice for multi-project development.
- Cross-platform: as long as your code is portable, Dune will be able to cross-compile it. Read more in the *Cross-Compilation* section.
- Release directly from any revision: Dune needs no setup stage. To release your project, simply point to a specific Git tag (named revision). Of course, you can add some release steps if you'd like, but it isn't necessary. For more information, please refer to [dune-release](#).

The first section below defines some terms used in this manual. The second section specifies the Dune metadata format, and the third one describes how to use the `dune` command.

#### 1.1.2 Terminology

**root**

The top-most directory in a GitHub repo, workspace, and project, differentiated by variables such as `#{workspace_root}` and `#{project_root}`. Dune builds things from this directory. It knows how to build

targets that are descendants of the root. Anything outside of the tree starting from the root is considered part of the *installed world*. Refer to *Finding the Root* to learn how the workspace root is determined.

### workspace

The subtree starting from each root. It can contain any number of projects that will be built simultaneously by Dune, and it must contain a `dune-workspace` file.

### project

A collection of source files that must include a `dune-project` file. It may also contain one or more packages. A project consists in a hierarchy of directories. Every directory (at the root, or a subdirectory) can contain a `dune` file that contains instructions to build files in that directory. Projects can be shared between different applications.

### package

A set of libraries and executables that opam builds and installs as one.

### installed world

Anything outside of the workspace. Dune doesn't know how to build things in the installed world.

### installation

The action of copying build artifacts or other files from the `<root>/_build` directory to the *installed world*.

### scope

Defined by any directory that contains at least one `<package>.opam` file. Typically, every project defines a single scope that is a subtree starting from this directory. Moreover, scopes are separate from your project's dependencies. The scope also determines where private items are visible. Private items include libraries or binaries that will not be installed. See *Dune Projects and Workspaces* for more details.

### build context

A specific configuration written in a `dune-workspace` file, which has a corresponding subdirectory in the `<root>/_build` directory. It contains all the workspace's build artifacts. Without this specific configuration from the user, there is always a `default` build context that corresponds to the executed Dune environment.

### build context root

The root of a build context named `foo` is `<root>/_build/<foo>`.

### build target

Specified on the command line, e.g., `dune build <target_path.exe>`. All targets that Dune knows how to build live in the `_build` directory.

### alias

A build target that doesn't produce any file and has configurable dependencies. Targets starting with `@` on the command line are interpreted as aliases (e.g., `dune build @src/runtest`). Aliases are per-directory. See *Aliases*.

### environment

Determines the default values of various parameters, such as the compilation flags. In Dune, each directory has an environment attached to it. Inside a scope, each directory inherits the environment from its parent. At the root of every scope, a default environment is used. At any point, the environment can be altered using an `env` stanza.

### build profile

A global setting that influences various defaults. It can be set from the command line using `--profile <profile>` or from `dune-workspace` files. The standard profiles are `dev` and `release`. `dev` is the default profile when none is set explicitly, and `release` is the profile used for opam releases. See *profile* for the exact defaults associated with these profiles.

### dialect

An alternative frontend to OCaml (such as ReasonML). It is described by a pair of file extensions, one corresponding to interfaces and one to implementations. It can use the standard OCaml syntax, or it can specify an action to convert from a custom syntax to a binary OCaml abstract syntax tree. It can also specify a custom formatter.

### placeholder substitution

A build step in which placeholders such as `%%VERSION%%` in source files are replaced by concrete values such as `1.2.3`. It is performed by `dune subst` for development versions and `dune-release` for releases.

### stanza

A fragment of a file interpreted by Dune, that will appear as a s-expression at the top-level of a file. For example, the `library` stanza describes a library. This can be either a generic term (“the library stanza”) or it can refer to a particular instance in a file (“the executable stanza in `bin/dune`”).

## 1.1.3 Project Layout

A typical Dune project will have a `dune-project` and one or more `<package>.opam` files at the root as well as `dune` files wherever interesting things are: libraries, executables, tests, documents to install, etc.

We recommend organising your project to have exactly one library per directory. You can have several executables in the same directory, as long as they share the same build configuration. If you’d like to have multiple executables with different configurations in the same directory, you will have to make an explicit module list for every executable using `modules`.

### 1.1.4 History

Dune started as `jbuilder` in late 2016. When its 1.0.0 version was released in 2018, the name has been changed to `dune`. It used to be configured with `jbuild` and `jbuild-workspace` files with a slightly different syntax. After a transition period, this syntax is not supported anymore.

## 1.2 Quickstart

This document gives simple usage examples of Dune. You can also look at [examples](#) for complete examples of projects using Dune with `CRAM` stanzas.

To try these examples, you will need to have Dune installed. See [How to Install Dune](#).

### 1.2.1 Initializing Projects

The following subsections illustrate basic usage of the `dune init proj` subcommand. For more documentation, see [Initializing Components](#) and the inline help available from `dune init --help`.

#### Initializing an Executable

To initialize a project that will build an executable program, run the following (replacing `project_name` with the name of your project):

```
$ dune init proj project_name
```

This creates a project directory that includes the following contents:

```
project_name/
├── dune-project
├── test
│   ├── dune
│   └── test_project_name.ml
├── lib
│   └── dune
├── bin
│   └── dune
```

(continues on next page)

(continued from previous page)

```
└─ main.ml
└─ project_name.opam
```

Now, enter your project's directory:

```
$ cd project_name
```

Then, you can build your project with:

```
$ dune build
```

You can run your tests with:

```
$ dune test
```

You can run your program with:

```
$ dune exec project_name
```

This simple project will print “Hello World” in your shell.

The following itemization of the generated content isn't necessary to review at this point. But whenever you are ready, it will provide jump-off points from which you can dive deeper into Dune's capabilities:

- The `dune-project` file specifies metadata about the project, including its name, packaging data (including dependencies), and information about the authors and maintainers. Open this in your editor to fill in the placeholder values. See [dune-project](#) for details.
- The `test` directory contains a skeleton for your project's tests. Add to the tests by editing `test/test_project_name.ml`. The generated tests can also use modules from `lib`; see the note about module names in the `bin` item below. See [Writing and Running Tests](#) for details on testing.
- The `lib` directory will hold the library you write to provide your executable's core functionality. Add modules to your library by creating new `.ml` files in this directory. See [library](#) for details on specifying libraries manually.
- The `bin` directory holds a skeleton for the executable program. Within the modules in this directory, you can access modules in your `lib` through the generated library's OCaml module name. For a project initialized as `project_name`, a source file such as `lib/foo.ml` is available as `Project_name.Foo` from both `bin` and `test`. In other words, `project_name` is not written literally in OCaml code: OCaml module names are capitalized. You can run the executable with `dune exec project_name`. See [Building a Hello World Program From Scratch](#) for an example of specifying an executable manually and [executable](#) for details.
- The `project_name.opam` file will be freshly generated from the `dune-project` file whenever you build your project. You shouldn't need to worry about this, but you can see [How Dune integrates with opam](#) for details.
- The `dune` files in each directory specify the component to be built with the files in that directory. For details on `dune` files, see [dune](#).

### Initializing a Library

To initialize a project for an OCaml library, run the following (replacing `project_name` with the name of your project):

```
$ dune init proj --kind=lib project_name
```

This creates a project directory that includes the following contents:

```

project_name/
├── dune-project
├── lib
│   └── dune
├── test
│   ├── dune
│   └── test_project_name.ml
└── project_name.opam

```

Now, enter your project's directory:

```
$ cd project_name
```

Then, you can build your project with:

```
$ dune build
```

You can run your tests with:

```
$ dune test
```

All of the subcomponents generated are the same as those described in *Initializing an Executable*, with the following exceptions:

- There is no `bin` directory generated.
- The `dune` file in the `lib` directory specifies that the library should be *public*. See *library* for details.

## 1.2.2 Building a Hello World Program From Scratch

Create a new directory within a Dune project (*Initializing an Executable*). Since OCaml is a compiled language, first create a `dune` file in Nano, Vim, or your preferred text editor. Declare the `hello_world` executable by including the following stanza (shown below). Name this initial file `dune` and save it.

```
(executable
 (name hello_world))
```

Create a second file containing the following code and name it `hello_world.ml` (including the `.ml` extension). It will implement the executable stanza in the `dune` file when built.

```
print_endline "Hello, world!"
```

Next, build your new program in a shell using this command:

```
$ dune build hello_world.exe
```

This will create a directory called `_build` and build the program: `_build/default/hello_world.exe`. Note that native code executables will have the `.exe` extension on all platforms (including non-Windows systems).

Finally, run it with the following command to see that it worked. In fact, the executable can both be built and run in a single step:

```
$ dune exec -- ./hello_world.exe
```

Voila! This should print “Hello, world!” in the command line.

### 1.2.3 Building a Hello World Program Using Lwt

Lwt is a concurrent library in OCaml.

In a directory of your choice, write this dune file:

```
(executable
 (name hello_world)
 (libraries lwt.unix))
```

This hello\_world.ml file:

```
Lwt_main.run (Lwt_io.printf "Hello, world!\n")
```

And build it with:

```
$ dune build hello_world.exe
```

The executable will be built as `_build/default/hello_world.exe`

### 1.2.4 Building a Hello World Program Using Core and Jane Street PPXs

Write this dune file:

```
(executable
 (name hello_world)
 (libraries core)
 (preprocess (pps ppx_jane)))
```

This hello\_world.ml file:

```
open Core

let () =
  Sexp.to_string_hum [%sexp ([3;4;5] : int list)]
  |> print_endline
```

And build it with:

```
$ dune build hello_world.exe
```

The executable will be built as `_build/default/hello_world.exe`

### 1.2.5 Defining a Library Using Lwt and ocaml-re

Write this dune file:

```
(library
 (name mylib)
 (public_name mylib)
 (libraries re lwt))
```

The library will be composed of all the modules in the same directory. Outside of the library, module `Foo` will be accessible as `Mylib.Foo`, unless you write an explicit `mylib.ml` file.

You can then use this library in any other directory by adding `mylib` to the `(libraries ...)` field.

## 1.2.6 Building a Hello World Program in Bytecode

In a directory of your choice, write this dune file:

```
;; This declares the hello_world executable implemented by hello_world.ml
;; to be build as native (.exe) or bytecode (.bc) version.
(executable
 (name hello_world)
 (modes byte exe))
```

This hello\_world.ml file:

```
print_endline "Hello, world!"
```

And build it with:

```
$ dune build hello_world.bc
```

The executable will be built as `_build/default/hello_world.bc`. If the executable stanza is in a subdirectory, include that path in the target. For example, an executable named `main` declared in `bin/dune` is built with `dune build bin/main.bc`.

The executable can be built and run in a single step with `dune exec ./hello_world.bc`. This bytecode version allows the usage of `ocamldebug`.

## 1.2.7 Setting the OCaml Compilation Flags Globally

Write this dune file at the root of your project:

```
(env
 (dev
  (flags (:standard -w +42)))
 (release
  (ocamlopt_flags (:standard -O3))))
```

`dev` and `release` correspond to build profiles. The build profile can be selected from the command line with `--profile foo` or from a `dune-workspace` file by writing:

```
(profile foo)
```

## 1.2.8 Using Cppo

Add this field to your library or executable stanzas:

```
(preprocess (action (run %{bin:cppo} -V OCAML:%{ocaml_version} %{input-file})))
```

Additionally, if you want to include a `config.h` file, you need to declare the dependency to this file via:

```
(preprocessor_deps config.h)
```

### Using the `.cppo.ml` Style Like the `ocamlbuild` Plugin

Write this in your dune file:

```
(rule
  (targets foo.ml)
  (deps (:first-dep foo.cppo.ml) <other files that foo.ml includes>)
  (action (run %{bin:cppo} %{first-dep} -o %{targets})))
```

### 1.2.9 Defining a Library with C Stubs

Assuming you have a file called `mystubs.c`, that you need to pass `-I/blah/include` to compile it and `-lblah` at link time, write this dune file:

```
(library
  (name mylib)
  (public_name mylib)
  (libraries re lwt)
  (foreign_stubs
    (language c)
    (names mystubs)
    (flags -I/blah/include))
  (c_library_flags (-lblah)))
```

### 1.2.10 Defining a Library with C Stubs using pkg-config

Same context as before, but using `pkg-config` to query the compilation and link flags. Write this dune file:

```
(library
  (name mylib)
  (public_name mylib)
  (libraries re lwt)
  (foreign_stubs
    (language c)
    (names mystubs)
    (flags (:include c_flags.sexp)))
  (c_library_flags (:include c_library_flags.sexp)))

(rule
  (targets c_flags.sexp c_library_flags.sexp)
  (action (run ./config/discover.exe)))
```

Then create a `config` subdirectory and write this dune file:

```
(executable
  (name discover)
  (libraries dune-configurator))
```

as well as this `discover.ml` file:

```
module C = Configurator.V1

let () =
  C.main ~name:"foo" (fun c ->
  let default : C.Pkg_config.package_conf =
    { libs = ["-lgst-editing-services-1.0"]
    ; cflags = []
```

(continues on next page)

(continued from previous page)

```

}
in
let conf =
  match C.Pkg_config.get c with
  | None -> default
  | Some pc ->
    match (C.Pkg_config.query pc ~package:"gst-editing-services-1.0") with
    | None -> default
    | Some deps -> deps
in

C.Flags.write_sexp "c_flags.sexp"      conf.cflags;
C.Flags.write_sexp "c_library_flags.sexp" conf.libs)

```

### 1.2.11 Using a Custom Code Generator

To generate a file `foo.ml` using a program from another directory:

```

(rule
  (targets foo.ml)
  (deps (:gen ../generator/gen.exe))
  (action (run %{gen} -o %{targets})))

```

### 1.2.12 Defining Tests

Write this in your dune file:

```
(test (name my_test_program))
```

And run the tests with:

```
$ dune runtest
```

It will run the test program (the main module is `my_test_program.ml`) and error if it exits with a nonzero code. See [Writing and Running Tests](#) for a more complete guide to testing with Dune.

In addition, if a `my_test_program.expected` file exists, it will be compared to the standard output of the test program and the differences will be displayed. It is possible to replace the `.expected` file with the last output using:

```
$ dune promote
```

### 1.2.13 Building a Custom Toplevel

A toplevel is simply an executable calling `Topmain.main ()` and linked with the compiler libraries and `-linkall`. Moreover, currently toplevels can only be built in bytecode.

As a result, write this in your dune file:

```

(executable
  (name      mytoplevel)
  (libraries compiler-libs.toplevel mylib)

```

(continues on next page)

(continued from previous page)

```
(link_flags (-linkall))  
(modes      byte))
```

And write this in `mytoplevel.ml`:

```
let () = exit (Topmain.main ())
```

## 1.3 Command-Line Interface

This section describes using `dune` from the shell.

### 1.3.1 Initializing Components

NOTE: The `dune init` command is still under development and subject to change.

Dune's `init` subcommand provides limited support for generating Dune file stanzas and folder structures to define components. The `dune init` command can be used to quickly add new projects, libraries, tests, and executables without having to manually create Dune files in a text editor, or it can be composed to programmatically generate parts of a multi-component project.

#### Initializing a Project

You can run the following command to initialize a new Dune project that uses the `base` and `cmdliner` libraries and supports inline tests:

```
$ dune init proj myproj --libs base,cmdliner --inline-tests --ppx ppx_inline_test
```

This creates a new directory called `myproj`, including subdirectories and `dune` files for library, executable, and test components. Each component's `dune` file will also include the declarations required for the given dependencies.

This is the quickest way to get a basic `dune` project up and building.

#### Initializing an Executable

To add a new executable to a `dune` file in the current directory (creating the file if necessary), run

```
$ dune init exe myexe --libs base,containers,notty --ppx ppx_deriving
```

This will add the following stanza to the `dune` file:

```
(executable  
  (name main)  
  (libraries base containers notty)  
  (preprocess  
    (pps ppx_deriving)))
```

#### Initializing a Library

Run the following command to create a new directory `src`, initialized as a library:

```
$ dune init lib mylib src --libs core --inline-tests --public
```

This will ensure the file `./src/dune` contains the below stanza (creating the file and directory, if necessary):

```
(library
  (public_name mylib)
  (inline_tests)
  (name mylib)
  (libraries core)
  (preprocess
    (pps ppx_inline_tests)))
```

### Initializing Components in a Specified Directory

All `init` subcommands take an optional `PATH` argument, which should be a path to a directory. When supplied, the component will be created in the specified directory. E.g., to initialize a project in the current working directory, run

```
$ dune init proj my_proj .
```

To initialize a project in a directory in some nested path, run

```
$ dune init proj my_proj path/to/my/project
```

If the specified directory does not already exist, it will be created.

### Learning More About the `init` Commands

Consult the manual page using the ``dune init --help`` command for more details.

### 1.3.2 Finding the Root

The root of the current workspace is determined by looking up a `dune-workspace` or `dune-project` file in the current directory and its parent directories. Dune requires at least one of these two files to operate.

If it isn't in the current directory and Dune produces output (such as errors or warnings), Dune prints `Entering directory` before the first message and `Leaving directory` at the end:

```
$ dune runtest
Entering directory '/home/jdimino/code/dune'
...
Leaving directory '/home/jdimino/code/dune'
```

If the build is silent (no errors, warnings, or other output), these messages are not printed. This message can also be suppressed with the `--no-print-directory` command line option (as in GNU `make`).

More precisely, Dune will choose the outermost ancestor directory containing a `dune-workspace` file, which is used to mark the root of the current workspace. If no `dune-workspace` file is present, the same strategy applies with `dune-project` files.

In case of a mix of *dune-workspace* and *dune-project* files, workspace files take precedence over project files in the sense that if a `dune-workspace` file is found, only parent `dune-workspace` files will be considered when looking for the root; however, if a *dune-project* file is found both parent `dune-workspace` and `dune-project` files will be considered.

A `dune-workspace` file is also a configuration file. Dune will read it unless the `--workspace` command line option is used. See *dune-workspace* for the syntax of this file. The scope of `dune-project` files is wider than the scope `dune-workspace` files. For instance, a `dune-project` file may specify the name of the project which is a universal property of the project, while a `dune-workspace` file may specify an `opam` switch name which is valid only on a given machine. For this reason, it is common and recommended to commit `dune-project` files in repositories, while it is less common to commit `dune-workspace` files.

### Current Directory

If the previous rule doesn't apply, i.e., no ancestor directory has a file named `dune-workspace`, then the current directory will be used as root.

### Forcing the Root (for Scripts)

You can pass the `--root` option to `dune` to select the root explicitly. This option is intended for scripts to disable the automatic lookup.

Note that when using the `--root` option, targets given on the command line will be interpreted relative to the given root, not relative to the current directory, as this is normally the case.

Alternatively, you can set the environment variable `DUNE_ROOT`. The `--root` option has a higher priority than `DUNE_ROOT` environment variable.

### 1.3.3 Interpretation of Targets

This section describes how Dune interprets the targets provided on the command line. When no targets are specified, Dune builds the *default alias*.

#### Resolution

All targets that Dune knows how to build live in the `_build` directory. Although, some are sometimes copied to the source tree for the need of external tools. These includes `<package>.install` files when either `-p` or `--promote-install-files` is passed on the command line.

As a result, if you want to ask Dune to produce a particular `.exe` file you would have to type:

```
$ dune build _build/default/bin/prog.exe
```

However, for convenience, when a target on the command line doesn't start with `_build`, Dune expands it to the corresponding target in all the build contexts that Dune knows how to build. When using `--verbose`, it prints out the actual set of targets upon starting:

```
$ dune build bin/prog.exe --verbose
...
Actual targets:
- _build/default/bin/prog.exe
- _build/4.03.0/bin/prog.exe
- _build/4.04.0/bin/prog.exe
```

If a target starts with the `@` sign, it is interpreted as an *alias*. See *Aliases*.

#### Variables for Artifacts

It's possible to build specific artifacts by using the corresponding variable on the command line. For example:

```
dune build '%{cmi:foo}'
```

See *Variables for Artifacts* for more information.

### 1.3.4 Finding External Libraries

When a library isn't available in the workspace, Dune will search for it in the installed world and expect it to be already compiled.

It looks up external libraries using a specific list of search paths, and each build context has a specific list of search paths.

When running inside an opam environment, Dune will look for installed libraries in `$OPAM_SWITCH_PREFIX/lib`. This includes both opam build context configured via the `dune-workspace` file and the default build context when the variable `$OPAM_SWITCH_PREFIX` is set.

Otherwise, Dune takes the directory where `ocamlc` was found and appends `./lib`` to it. For instance, if `ocamlc` is found in `/usr/bin`, Dune looks for installed libraries in `/usr/lib`.

In addition to the two above rules, Dune always inspects the `OCAMLPATH` environment variable and uses the paths defined in this variable. `OCAMLPATH` always has precedence and can have different values in different build contexts. For instance, you can set it manually in a specific build context via the `dune-workspace` file.

### 1.3.5 Running Tests

There are two ways to run tests:

- `dune build @runtest`
- `dune test` (or its alias `dune runtest`)

For projects with only the default build context, both commands run all the tests defined in the current directory and its children directories recursively. They differ when a workspace defines multiple build contexts: `dune build @runtest` builds the alias as a normal build target, while `dune test` selects one context for source paths. See [Running Tests](#) for details.

You can also run the tests in a specific sub-directory and its children by using:

- `dune build @foo/bar/runtest`
- `dune test foo/bar` (or `dune runtest foo/bar`)

### 1.3.6 Watch Mode

The `dune build` and `dune runtest` commands support a `-w` (or `--watch`) flag. When it's passed, Dune will perform the action as usual and then wait for file changes and rebuild (or rerun the tests). This feature requires `inotifywait` or `fswatch` to be installed.

To work with the watch mode, it is recommended to create a `dune` file in `start` (purely by convention) and define a particular alias in it, such as `build` (again, any name can be used). For example:

```
$ dune build --watch -alias start/build
```

Now, build requests are just dependencies of this alias specified in `start/dune`:

```
(alias
 (name build)
 (deps (alias_rec %{project_root}/runtest)))
```

As this file is edited, dune will pick up the build requests in it.

It is customary to add this file to `.gitignore`.

This pattern is common that `$ dune init start-file` has been introduced to help create such start files

### 1.3.7 Launching the Toplevel (REPL)

Dune supports launching a `utop` instance with locally defined libraries loaded.

```
$ dune utop <dir> -- <args>
```

Where `<dir>` is a directory under which Dune searches (recursively) for all libraries that will be loaded. `<args>` will be passed as arguments to the `utop` command itself. For example, `dune utop lib -- -implicit-bindings` will start `utop`, with the libraries defined in `lib` and implicit bindings for toplevel expressions.

Dune also supports loading individual modules unsealed by their signatures into the toplevel. This is accomplished by launching a toplevel and then asking dune to return the toplevel directives needed to evaluate the module:

```
$ utop
# use_output "dune ocaml top-module path/to/module.ml";;
```

#### Requirements & Limitations

- Utop version `>= 2.0` is required for this to work.
- This subcommand only supports loading libraries. Executables aren't supported.
- Libraries that are dependencies of `utop` itself cannot be loaded. For example `Camomile`.
- Loading libraries that are defined in different directories into one `utop` instance isn't possible.

### 1.3.8 Restricting the Set of Packages

Restrict the set of packages from your workspace that Dune can see with the `--only-packages` option:

```
$ dune build --only-packages pkg1,pkg2,... @install
```

This option acts as if you went through all the Dune files and commented out the stanzas referring to a package that isn't in the list given to dune.

### 1.3.9 Distributing Projects

Dune provides support for building and installing your project; however, it doesn't provide helpers for distributing it. It's recommended to use `dune-release` for this purpose.

The common defaults are that your projects include the following files:

- `README.md`
- `CHANGES.md`
- `LICENSE.md`

If your project contains several packages, all the package names must be prefixed by the shortest one.

#### 1.3.10 `dune subst`

One of the features `dune-release` provides is watermarking; it replaces various strings of the form `%ID%` in all your project files before creating a release tarball or when the opam user pins the package.

This is especially interesting for the `VERSION` watermark, which gets replaced by the version obtained from the Version-Control System (VCS). For instance, if you're using Git, `dune-release` invokes this command to find out the version:

```
$ git describe --always --dirty --abbrev=7
1.0+beta9-79-g29e9b37
```

If no VCS is detected, `dune subst` will do nothing.

Projects using Dune usually only need `dune-release` for creating and publishing releases. However, they may still substitute the watermarks when the user pins the package. To help with this, Dune provides the `subst` sub-command.

`dune subst` performs the same substitution that `dune-release` does with the default configuration, i.e., calling `dune subst` at the root of your project will rewrite all your project files.

More precisely, it replaces the following watermarks in the source files:

- `NAME`, the name of the project
- `VERSION`, output of `git describe --always --dirty --abbrev=7`
- `VERSION_NUM`, same as `VERSION` but with a potential leading `v` or `V` dropped
- `VCS_COMMIT_ID`, commit hash from the vcs
- `PKG_MAINTAINER`, contents of the `maintainer` field from the opam file
- `PKG_AUTHORS`, contents of the `authors` field from the opam file
- `PKG_HOMEPAGE`, contents of the `homepage` field from the opam file
- `PKG_ISSUES`, contents of the `issues` field from the opam file
- `PKG_DOC`, contents of the `doc` field from the opam file
- `PKG_LICENSE`, contents of the `license` field from the opam file
- `PKG_REPO`, contents of the `repo` field from the opam file

The project name is obtained by reading the `dune-project` file in the directory where `dune subst` is called. The `dune-project` file must exist and contain a valid `(name ...)` field.

Note that `dune subst` is meant to be called from the opam file and behaves a bit different to other Dune commands. In particular it doesn't try to detect the root of the workspace and must be called from the root of the project.

### 1.3.11 Custom Build Directory

By default Dune places all build artifacts in the `_build` directory relative to the user's workspace. However, one can customize this directory by using the `--build-dir` flag or the `DUNE_BUILD_DIR` environment variable.

```
$ dune build --build-dir _build-foo

# this is equivalent to:
$ DUNE_BUILD_DIR=_build-foo dune build

# Absolute paths are also allowed
$ dune build --build-dir /tmp/build foo.exe
```

### 1.3.12 Controlling Concurrency

By default Dune automatically detects the number of CPU cores and runs that many jobs in parallel. You can override this using the `-j` flag or the `DUNE_JOBS` environment variable with either a positive integer or `auto` to use the default auto-detection.

```
$ dune build -j 4

# this is equivalent to:
$ DUNE_JOBS=4 dune build
```

The command-line option takes precedence over the environment variable, which takes precedence over the *jobs* setting in the configuration file.

### 1.3.13 Installing a Package

#### Via opam

When releasing a package using Dune in opam, there's nothing special to do. Dune generates a file called `<package-name>.install` at the root of the project. This contains a list of files to install, and opam reads it in order to perform the installation.

#### Manually

When not using opam, or when you want to manually install a package, you can ask Dune to perform the installation via the `install` command:

```
$ dune install [PACKAGE]...
```

This command takes a list of package names to install. If no packages are specified, Dune will install all available packages in the workspace. When several build contexts are specified via a *dune-workspace* file, Dune performs the installation in all the build contexts.

#### Destination Directory

For a given build context, the installation directories are determined with a single scheme for all installation sections. Taking the `lib` installation section as an example, the priorities of this scheme are as follows:

1. if an explicit `--libdir <path>` argument is passed, use this path
2. if an explicit `--prefix <path>` argument is passed, use `<path>/lib`
3. if a `--libdir <path>` argument was passed during Dune's compilation to `./configure`, use this path
4. if `OPAM_SWITCH_PREFIX` is present in the environment, use `$OPAM_SWITCH_PREFIX/lib`
5. otherwise, fail

The same scheme applies to every section, each with its own flag: `--bindir`, `--sbindir`, `--libexecdir`, `--datadir` (the `share` section), `--docdir`, `--etcdir`, and `--mandir`. The directories derived from `--prefix` follow the opam layout: in particular `man` becomes `<prefix>/man` and `doc` becomes `<prefix>/doc` – they are *not* placed under `<prefix>/share`.

Note that `--prefix` (priority 2 above) takes precedence over a directory configured at build time via `./configure` (priority 3). Passing `--prefix` therefore overrides a `--mandir` (or `--docdir`, etc.) that was set at configure time. A system installation following the [Filesystem Hierarchy Standard](#), which expects `<prefix>/share/man` and `<prefix>/share/doc`, should pass the per-section flags explicitly:

```
$ dune install --prefix /usr --mandir /usr/share/man --docdir /usr/share/doc PACKAGE
```

#### Relocation Mode

The installation can be done in specific mode (`--relocation`) for creating a directory that can be moved. In that case, the installed executables will look up the package sites (cf *How to Load Additional Files at Runtime*) relative to its location. The `-prefix` directory should be used to specify the destination.

If you're using plugins that depend on installed libraries and aren't executable dependencies, like libraries that need to be loaded at runtime, you must copy the libraries manually to the destination directory.

### 1.3.14 Querying Merlin Configuration

Since Version 2.8, Dune no longer promotes `.merlin` files to the source directories. Instead, Dune stores these configurations in the `_build` folder, and Merlin communicates directly with Dune to obtain its configuration via the `ocaml-merlin` subcommand. The Merlin configuration is now stanza-specific, allowing finer control. The following commands aren't needed for normal Dune and Merlin use, but they can provide insightful information when debugging or configuring non-standard projects.

#### Printing the Configuration

It's possible to manually query the generated configuration for debugging purposes:

```
$ dune ocaml merlin dump-config
```

This command prints the distinct configuration of each module present in the current directory. This directory must be in a Dune workspace and the project must be already built. The configuration will be encoded as s-expressions, which are used to communicate with Merlin.

#### Printing an Approximated `.merlin`

It's also possible to print the current folder's configuration in the Merlin configuration syntax by running the following command:

```
$ dune ocaml dump-dot-merlin > .merlin
```

In that case, Dune prints only one configuration: the result of the configuration's coarse merge in the current folder's various modules. This folder must be in a Dune workspace, and the project must be already built. Preprocessing directives and other flags will be commented out and must be un-commented afterward. This feature doesn't aim at writing exact or correct `.merlin` files. Its sole purpose is to lessen the burden of writing the configuration from scratch.

#### Non-Standard Filenames

Merlin configuration loading is based on filenames, so if you have files that are preprocessed by custom rules before they are built, they should respect the following naming convention: the unprocessed file should start with the name of the resulting processed file followed by a dot. The rest does not matter. Dune uses only the name before the first dot to match with available configurations.

For example, if you use the `cppo` preprocessor to generate the file `real_module_name.ml`, then the source file could be named `real_module_name.cppo.ml`.

### 1.3.15 Running a Rocq Toplevel

See *Running a Rocq Toplevel*.



These tutorials are hands-on lessons to learn about Dune.

## 2.1 Developing with Dune

### Warning

This tutorial is a work in progress.

In this tutorial, will start with a small Dune project, and extend it by using common features:

- the *executable*, *library*, and *test stanzas*;
- *cram* tests;
- bindings to C code using *foreign stubs*;
- using a *ppx deriver*.

By doing so, you'll interact with `dune runtest` and `dune promote`, and will use the most common *stanzas* in Dune files.

Start the tutorial with the *Introduction*.

### 2.1.1 Introduction

The goal of this first chapter is to get to a place where you have a working Dune project with a skeleton of a calculator.

This is a **tutorial**: it is meant to be followed in order, but you can stop at any point. You can also restart from any chapter, using these sections that are present at the end of the previous chapter.

#### Checkpoint

This will contain the project at the end of each chapter.

#### Installing Packages

First, you'll need to have a working Opam installation. This is described in *How to Install Dune*.

Then, create an empty directory somewhere, say `~/dune-calc`. In this tutorial, we will only create files in this directory.

Let's first make sure you have a working opam installation. Run this command:

```
opam --version
```

It should display something like “2.2.0”. Anything greater than 2.0.0 is fine.

### Important

When asked to type a command, you can click to copy the full command to your clipboard.

In this tutorial, all commands will be typed at the root of your project, like `~/dune-calc`.

Let’s create a local switch: `cd` to this directory and run the following command. This can take a few minutes.

```
opam switch create ./ 5.4.0
```

This command has created a directory named `_opam` in the current directory. Now, let’s install some packages by running:

```
opam install dune.3.15.3 menhir.20231231
```

You can confirm that `opam` is correctly setup by typing `dune --version`, which should display 3.15.3. Otherwise, please refer to *How to Install Dune*.

### Note

The instructions use precise version numbers in `opam install` command. This is to ensure that the error messages will exactly map what you’re seeing, but it is very likely to work with any version.

## The Calculator Skeleton

Now that we have an `opam` switch and some packages installed, let’s create the various files.

For each file, click to reveal the file contents and click to copy the contents to your clipboard. Open `file.txt` in a text editor and paste the contents there.

### `dune-project`

```
(lang dune 3.18)
(using menhir 3.0)
(package (name calc))
```

This file contains metadata about the project:

- the version of the dune language we’re using
- the extensions we’re *using*
- the *package* we’re defining

### See also

*dune-project* Reference documentation about `dune-project` files

**dune**

```
(executable
  (public_name calc))

(ocamllex lexer)

(menhir
  (modules parser))
```

This file contains a description of what's in our project:

- an *executable* stanza defining our calculator binary
- an *ocamllex* stanza, setting up rules to compile `lexer.mll` to a Lexer module
- a *menhir* stanza, to similarly use `parser.mly` as a Parser module

**parser.mly**

```
%token Eof
%token<int> Int
%token Plus
%start<Ast.exp> main

%left Plus

%{ open Ast %}

%%

main: expr Eof { $1 }

expr:
| Int { Int $1 }
| expr Plus expr { Add ($1, $3) }

%%
```

This contains definitions of our tokens and grammar rules, using *Menhir*.

**lexer.mll**

```
let space = [' ']+
let digit = ['0'-'9']

rule token = parse
  | eof { Parser.Eof }
  | space { token lexbuf }
  | '\n' { Parser.Eof }
  | '+' { Parser.Plus }
  | digit+ { Parser.Int (int_of_string (Lexing.lexeme lexbuf)) }
```

This is our lexer, using *ocamllex*.

### ast.ml

```
type exp =  
  | Int of int  
  | Add of exp * exp
```

This contains a definition of the arithmetic expressions manipulated by the calculator.

#### Note

This is in a separate file from `calc.ml` to avoid module cycles, since `Calc` depends on `Parser`, which depends on the expression type.

### calc.ml

```
let rec eval = function Ast.Int n -> n | Add (a, b) -> eval a + eval b  
  
let () =  
  while true do  
    Printf.printf ">> %!";  
    let lb = Lexing.from_channel Stdlib.stdin in  
    let e = Parser.main Lexer.token lb in  
    Printf.printf "%d\n" (eval e)  
  done
```

This is the “business logic” of our app, in which we:

- display a prompt
- call the lexer and parser to get an expression
- evaluate the expression
- display the result

At this stage, we have the skeleton of a calculator.

Run the following command to build and execute the calculator:

```
dune exec calc
```

You can enter additions, such as `1+2` followed by `Enter`. Exit with `Ctrl+C`.

Initially, only addition is supported and anything else triggers an exception terminating the execution.

Note that a `_build` directory is now present. This is where Dune will store all compiled artifacts.

You can safely remove this directory - that’s actually what the `dune clean` command does. But that’s not usually necessary since Dune will keep track of dependencies and what is up to date.

The `_opam` directory is where your dependencies are located. It is managed by `opam`. If it gets removed by accident or something is corrupted in there, it is safe to remove it and recreate it by running `opam switch` and `opam install` as described above.

## 2.1.2 Improving Structure

Our calculator is fairly monolithic at this stage. Instead of a single executable, we're going to extract a library and create some tests. For now, this is just a cram test that will call the executable, but this structure will later allow adding unit tests for the library.

### Extract a Library

Create folders `bin`, `lib` and `test`, for binary, library, and tests, respectively.

Run the following command to install `cmdliner`:

```
opam install cmdliner.1.3.0
```

Let's create a library. Create `lib/dune` with the `(ocamllex)` and `(menhir)` stanzas from the original `dune` file and a new `(library)` stanza:

```
(library
 (name calc)
 (libraries cmdliner))

(ocamllex lexer)

(menhir
 (modules parser))
```

#### Note

This is the whole contents of the file. The `(library)` part is highlighted to show that it's the part that we've just added.

#### Note

We're defining a *library* that depends on the `cmdliner` library.

Libraries can either be defined in your project, or provided by an opam package. In the case of `cmdliner`, this is the latter, since we've installed it just before.

*The OCaml Ecosystem* covers the difference between packages, libraries, and modules.

Move `ast.ml`, `lexer.mll`, and `parser.mly` to the `lib` directory.

Now we're going to move `calc.ml` to `lib/cli.ml` and replace it by the following:

```
let rec eval = function Ast.Int n -> n | Add (a, b) -> eval a + eval b

let info = Cmdliner.Cmd.info "calc"

let eval_lb lb =
  let e = Parser.main Lexer.token lb in
  Printf.printf "%d\n" (eval e)

let repl () =
  while true do
```

(continues on next page)

(continued from previous page)

```

Printf.printf ">> %!";
let lb = Lexing.from_channel Stdlib.stdin in
eval_lb lb
done

let term =
  let open Cmdliner.Term.Syntax in
  let+ expr_opt =
    let open Cmdliner.Arg in
    value & opt (some string) None & info [ "e" ]
  in
  match expr_opt with
  | Some s -> eval_lb (Lexing.from_string s)
  | None -> repl ()

let cmd = Cmdliner.Cmd.v info term
let main () = Cmdliner.Cmd.eval cmd |> Stdlib.exit

```

**Note**

Two things are happening here.

We are adding a second code path to evaluate a `string` directly, so we extract an `eval_lb` function that operates on a `lexbuf` (the “source” a lexer can read from).

We are also moving to `cmdliner` for command-line parsing. This consists in:

- an `info` value (of type `Cmdliner.Cmd.info`) which contains metadata for the program (used in help, etc)
- a `term` value (of type `unit Cmdliner.Term.t`) which sets up arguments and calls `eval_lb` with the right `lexbuf`
- a `cmd` value (of type `unit Cmdliner.Cmd.t`) grouping `info` and `term` together
- a `main` function of type `unit -> 'a` to run `cmd`

**Extract an Executable**

Let’s create an executable in `bin`. To do so, create a `bin/dune` file with the following contents:

```

(executable
 (public_name calc)
 (libraries calc))

```

And `bin/calc.ml` with a single function call:

```

let () = Calc.Cli.main ()

```

Delete the `dune` at the root.

## Create a Test

Create `test/calc.t` with the following contents.

### Important

In *cram tests*, commands start with two spaces, a dollar sign, and a space.  
Make sure to include **two spaces** at the beginning of the line.

```
$ calc -e '1+2'
```

Now create `test/dune` to inform Dune that `cram` tests will use our `calc` executable and need to be executed again when it changes:

```
(cram
 (deps %bin:calc))
```

At this stage, we're ready to run our test.

Let's do this with `dune runtest`.

It's displaying a diff:

```
$ calc -e '1+2'
+ 3
```

Now, run `dune promote`. The contents of `test/calc.t` have changed. Most editors will pick this up automatically, but it might be necessary to reload the file to see the change.

Finally, run `dune runtest`. Nothing happens.

Now, run the calculator by running `dune exec calc` to confirm that the interactive mode still works.

### Note

What happened here? This Dune feature, where some tests can edit the source file, is called *promotion*.

*Cram tests* contain both commands and their expected input. We did not include any output in the initial `cram` test. When running `dune runtest` for the first time, Dune executes the commands, and calls `diff` between the *expected output* (in `test/calc.t`: no output at all) and the *actual output* (from running the command: the line "3"), which will display added lines with a + sign and deleted lines with a - sign.

Running `dune promote` replaces the input file (`test/calc.t`) with the last *actual output*. So this includes the line with "3".

Running `dune runtest` again will execute the test again and compare the *expected output* (`test/calc.t` with the "3" line in it) with the *actual output* and finds no difference. This means that the test passes.

## Checkpoint

This is how the project looks like at the end of this chapter.

Listing 1: dune-project (unchanged)

```
(lang dune 3.18)
(using menhir 3.0)
(package (name calc))
```

Listing 2: bin/dune

```
(executable
 (public_name calc)
 (libraries calc))
```

Listing 3: bin/calc.ml

```
let () = Calc.Cli.main ()
```

Listing 4: lib/dune

```
(library
 (name calc)
 (libraries cmdliner))

(ocamllex lexer)

(menhir
 (modules parser))
```

Listing 5: lib/ast.ml (unchanged)

```
type exp =
 | Int of int
 | Add of exp * exp
```

Listing 6: lib/cli.ml

```
let rec eval = function Ast.Int n -> n | Add (a, b) -> eval a + eval b

let info = Cmdliner.Cmd.info "calc"

let eval_lb lb =
  let e = Parser.main Lexer.token lb in
  Printf.printf "%d\n" (eval e)

let repl () =
  while true do
    Printf.printf ">> %!";
    let lb = Lexing.from_channel Stdlib.stdin in
    eval_lb lb
  done

let term =
  let open Cmdliner.Term.Syntax in
  let+ expr_opt =
```

(continues on next page)

(continued from previous page)

```

let open Cmdliner.Arg in
  value & opt (some string) None & info [ "e" ]
in
match expr_opt with
| Some s -> eval_lb (Lexing.from_string s)
| None -> repl ()

let cmd = Cmdliner.Cmd.v info term
let main () = Cmdliner.Cmd.eval cmd |> Stdlib.exit

```

Listing 7: lib/lexer.mll (unchanged)

```

let space = [' ']+
let digit = ['0'-'9']

rule token = parse
  | eof { Parser.Eof }
  | space { token lexbuf }
  | '\n' { Parser.Eof }
  | '+' { Parser.Plus }
  | digit+ { Parser.Int (int_of_string (Lexing.lexeme lexbuf)) }

```

Listing 8: lib/parser.mly (unchanged)

```

%token Eof
%token<int> Int
%token Plus
%start<Ast.exp> main

%left Plus

%{ open Ast %}

%%

main: expr Eof { $1 }

expr:
| Int { Int $1 }
| expr Plus expr { Add ($1, $3) }

%%

```

Listing 9: test/dune

```

(cram
  (deps %{bin:calc}))

```

Listing 10: test/calc.t

```
$ calc -e '1+2'  
3
```

### 2.1.3 The Development Cycle

Our calculator now has a better structure, but it still does not do much.

In this chapter, we are going to make several iterations where we add a cram test, see it fail, implement the missing feature, and repeat. This is how many OCaml projects are developed (including Dune itself).

#### Note

There are many ways to do this: tests can be written first or last; some prefer to **promote** only the fixed version.

In this tutorial, we're going to write tests first and **promote** the erroneous version but as you become more familiar with Dune, you'll be able to explore variations of this loop.

#### Display Errors

So far, our calculator is not doing any error handling.

We're going to create a test with an error, see how the calculator behaves, and add a better error message.

#### Create a Test

Add a new test in `test/calc.t` with the following content. Note that we do not specify any output for the command. As before, make sure to include two spaces before the `$` sign.

```
$ calc -e '1+2'  
3  
  
$ calc -e '1+'
```

Next, run the tests using `dune runtest`. Dune will display a diff with the actual output:

```
$ calc -e '1+'  
+ calc: internal error, uncaught exception:  
+   Calc.Parser.MenhirBasics.Error  
+  
+ [125]
```

Run `dune promote` and observe that the error message is part of the test. At that point, running `dune runtest` will succeed.

Our goal for the rest of this section is to make this test display a nice error message.

#### Handle the Exception

The message points to an uncaught exception. Indeed, `Parser.main` can raise `Parser.Error`. Let's catch this exception in `eval_lb` and display the location of the error in the input file.

Edit `lib/cli.ml`:

```
let eval_lb lb =
  try
    let e = Parser.main Lexer.token lb in
    Printf.printf "%d\n" (eval e)
  with Parser.Error ->
    Printf.printf "parse error near character %d" lb.lex_curr_pos
```

**Note**

This is just an excerpt from the file. Edit the `eval_lb` to make it look like this. The modified parts are highlighted.

Now, run the tests again with `dune runtest`. It displays the following diff:

```
$ calc -e '1+'
- calc: internal error, uncaught exception:
-   Calc.Parser.MenhirBasics.Error
-
- [125]
+ parse error near character 2
```

Run `dune promote` and note that `test/calc.t` has changed.

Run `dune runtest`. Nothing is displayed, indicated that the test has passed.

**Note**

This is similar to what happened at the end of *the previous chapter*.

The first `dune runtest` compares the *expected output* (from `test/calc.t`, the uncaught exception message) to the *actual output* (our new error message) and displays the diff. So, the uncaught exception appears as deleted lines (prefixed with `-`) and the new error message appears as added lines (prefixed with `+`).

Running `dune promote` will copy the last *actual output* to `calc/test.t`.

Running `dune runtest` a second time will compare the *expected output* (the new message) with the *actual output* of the command (the new message) and find no difference.

**Add Floats**

At this stage, our calculator only supports integers. In this section, we are going to add support for floating-point numbers and operations.

**Add a Test**

First, add a test at the end of `test/calc.t`:

```
$ calc -e '1+2.5'
```

Run `dune runtest`: this displays an error.

```
$ calc -e '1+2.5'
+ calc: internal error, uncaught exception:
+   Failure("lexing: empty token")
```

(continues on next page)



(continued from previous page)

Here `is` an example of a case that `is not` matched:  
`Float _`

To fix this, we need to tweak `lib/cli.ml` a bit. Instead of returning an `int`, let's introduce a value type that can represent either `int` or `float`:

```
type value = VInt of int | VFloat of float

let value_to_string = function
| VInt n -> string_of_int n
| VFloat f -> Printf.sprintf "%.6g" f
```

And update the `eval_lb` function to use this printer:

```
let eval_lb lb =
  try
    let expr = Parser.main Lexer.token lb in
    let v = eval expr in
    Printf.printf "%s\n" (value_to_string v)
  with Parser.Error ->
    Printf.printf "parse error near character %d" lb.lex_curr_pos
```

Finally, for `eval` itself, we'll use `(+)` or `(+.)` if both values have the same type, or convert integers to floats if needed:

```
let rec eval = function
| Ast.Int n -> VInt n
| Float f -> VFloat f
| Add (a, b) -> (
  match (eval a, eval b) with
  | VInt na, VInt nb -> VInt (na + nb)
  | VFloat fa, VFloat fb -> VFloat (fa +. fb)
  | VInt na, VFloat fb -> VFloat (float na +. fb)
  | VFloat fa, VInt nb -> VFloat (fa +. float nb))
```

With this implementation done, call `dune runtest` and notice that it changes the output. Call `dune promote` to update the test.

## Pi

In this section, we're going to add named constants, and `pi` in particular.

### Create a test

Add a new test in `test/calc.t`:

```
$ calc -e '1+pi'
```

Run `dune runtest` and see the failure. Run `dune promote` to add the failure to the test file. Our goal in the rest of the section is to change the output of this test.

### Add a Constructor

Let's add a new constructor in `lib/ast.ml`:

```
type exp =  
  | Int of int  
  | Add of exp * exp  
  | Float of float  
  | Ident of string
```

That way, `pi` is going to be represented as `Ident "pi"`.

### Lexing and Parsing

We now need to parse these constants.

Let's add a new token in `lib/parser.mly`:

```
%token Eof  
%token<int> Int  
%token<float> Float  
%token Plus  
%token<string> Ident
```

Produce it using a new lexing rule in `lib/lexer.mll`:

```
let digit = ['0'-'9']  
let letter = ['a'-'z']  
let ident = letter+  
  
rule token = parse  
  | eof { Parser.Eof }  
  | space { token lexbuf }  
  | '\n' { Parser.Eof }  
  | '+' { Parser.Plus }  
  | digit+ { Parser.Int (int_of_string (Lexing.lexeme lexbuf)) }  
  | digit+ '.' digit+ { Parser.Float (float_of_string (Lexing.lexeme lexbuf)) }  
  | ident { Parser.Ident (Lexing.lexeme lexbuf) }
```

And handle it as a new derivation in `lib/parser.mly`:

```
expr:  
  | Int { Int $1 }  
  | Float { Float $1 }  
  | expr Plus expr { Add ($1, $3) }  
  | Ident { Ident $1 }
```

### Evaluation

Finally, we'll have to update our evaluation function to take the new constructor into account.

OCaml does not have a value for `pi`, but it has a `Stdlib.acos` function. Since  $\cos \frac{\pi}{2} = 0$ , we can define `pi` as `2 * arccos 0`.

```

let rec eval = function
| Ast.Int n -> VInt n
| Float f -> VFloat f
| Add (a, b) -> (
  match (eval a, eval b) with
  | VInt na, VInt nb -> VInt (na + nb)
  | VFloat fa, VFloat fb -> VFloat (fa +. fb)
  | VInt na, VFloat fb -> VFloat (float na +. fb)
  | VFloat fa, VInt nb -> VFloat (fa +. float nb))
| Ident "pi" -> VFloat (2. *. Stdlib.acos 0.)
| Ident _ -> failwith "unknown ident"

```

Finally, let's run our test. `dune runtest` will display a diff with the new value. Run `dune promote` to accept it.

## Multiplication

### Create a Test

Let's add a new test in `test/calc.t`:

```
$ calc -e '1+2*3'
```

Run the test with `dune runtest`. Notice the error message.

Let's run `dune promote` to add it to the file.

Now, our goal for the rest of this section is to change that to the expected result.

### Add a Constructor

Let's update the AST `lib/ast.ml`: we're generalizing addition to binary operations, and create a new `Mul` operation (notice that we remove the line corresponding to the `Add` expression).

```

type op =
| Add
| Mul

type exp =
| Int of int
| Float of float
| Ident of string
| Op of op * exp * exp

```

## Lexing and Parsing

Let's add a new token for `*` in `lib/parser.mly`:

```

%token<string> Ident
%token Plus
%token Star

```

Then, we'll produce it using a new rule in `lib/lexer.mll`:

```
| space { token lexbuf }  
| '\n' { Parser.Eof }  
| '+' { Parser.Plus }  
| '*' { Parser.Star }
```

And use that token in a new rule in `lib/parser.mly` (also modifying the `Add` rule):

```
| Int { Int $1 }  
| Float { Float $1 }  
| Ident { Ident $1 }  
| expr Plus expr { Op (Add, $1, $3) }  
| expr Star expr { Op (Mul, $1, $3) }
```

We have a last edit to do here, which is to add a precedence annotation:

```
%left Plus  
%left Star
```

## Evaluation

Now, we can update our evaluation function in `lib/cli.ml`:

```
let eval_number_op f_int f_float va vb =  
  match (va, vb) with  
  | VInt na, VInt nb -> VInt (f_int na nb)  
  | VFloat fa, VFloat fb -> VFloat (f_float fa fb)  
  | VInt na, VFloat fb -> VFloat (f_float (float_of_int na) fb)  
  | VFloat fa, VInt nb -> VFloat (f_float fa (float_of_int nb))  
  
let rec eval = function  
| Ast.Int n -> VInt n  
| Float f -> VFloat f  
| Ident "pi" -> VFloat (2. *. Stdlib.acos 0.)  
| Ident _ -> failwith "unknown ident"  
| Op (Add, a, b) -> eval_number_op ( + ) ( +. ) (eval a) (eval b)  
| Op (Mul, a, b) -> eval_number_op ( * ) ( *. ) (eval a) (eval b)
```

With these updates done, let's run `dune runtest`. It displays that the result is 7. Call `dune promote` to update the test.

## Division

This section is going to be very similar to the previous one. Instead we're adding the division operator.

### Create a Test

Add a test in `test/calc.t`:

```
$ calc -e '4/2'
```

Call `dune runtest`, note the error, call `dune promote`.

## Add a Constructor

Add a constructor in `lib/ast.ml`:

```
type op =
| Add
| Mul
| Div
```

## Lexing and Parsing

Update `lib/parser.mly`:

```
%token<string> Ident;
%token Plus
%token Star
%token Slash
```

Then add the right precedence:

```
%left Plus
+%left Star Slash
```

And the corresponding rule:

```
| Ident { Ident $1 }
| expr Plus expr { Op (Add, $1, $3) }
| expr Star expr { Op (Mul, $1, $3) }
| expr Slash expr { Op (Div, $1, $3) }
```

Finally, add a lexing rule in `lib/lexer.mll`:

```
| '+' { Parser.Plus }
| '*' { Parser.Star }
| '/' { Parser.Slash }
```

## Evaluation

Add a new case in `lib/cli.ml`:

```
let rec eval = function
| Ast.Int n -> VInt n
| Float f -> VFloat f
| Ident "pi" -> VFloat (2. *. Stdlib.acos 0.)
| Ident _ -> failwith "unknown ident"
| Op (Add, a, b) -> eval_number_op ( + ) ( +. ) (eval a) (eval b)
| Op (Mul, a, b) -> eval_number_op ( * ) ( *. ) (eval a) (eval b)
| Op (Div, a, b) -> eval_number_op ( / ) ( /. ) (eval a) (eval b)
```

Now, running `dune runtest` should display the right result. Run `dune promote` to accept it.

### Sine

#### Create a Test

Create a new test in `test/calc.t`:

```
$ calc -e 'sin (pi / 6)'
```

Call `dune runtest`, note the error, call `dune promote`.

#### Add a Constructor

Add a constructor in `lib/ast.ml`:

```
type exp =  
  | Int of int  
  | Float of float  
  | Ident of string  
  | Op of op * exp * exp  
  | Call of string * exp
```

### Lexing and Parsing

Update `lib/parser.mly` by defining new tokens for parentheses:

```
%token<string> Ident;  
%token Plus  
%token Star  
%token Slash  
%token Lpar Rpar
```

And a production for calls:

```
| Ident { Ident $1 }  
| expr Plus expr { Op (Add, $1, $3) }  
| expr Star expr { Op (Mul, $1, $3) }  
| expr Slash expr { Op (Div, $1, $3) }  
| Ident Lpar expr Rpar { Call ($1, $3) }
```

Update `lib/lexer.mll`:

```
| '+' { Parser.Plus }  
| '*' { Parser.Star }  
| '/' { Parser.Slash }  
| '(' { Parser.Lpar }  
| ')' { Parser.Rpar }
```

### Evaluation

We'll need a float to pass to `Stdlib.sin`, so let's introduce a conversion function. We'll also add the corresponding cases in `lib/cli.ml`:

```
let as_float = function  
  | VInt n -> float_of_int n
```

(continues on next page)

(continued from previous page)

```
| VFloat f -> f

let rec eval = function
| Ast.Int n -> VInt n
| Float f -> VFloat f
| Ident "pi" -> VFloat (2. *. Stdlib.acos 0.)
| Ident _ -> failwith "unknown ident"
| Op (Add, a, b) -> eval_number_op ( + ) ( +. ) (eval a) (eval b)
| Op (Mul, a, b) -> eval_number_op ( * ) ( *. ) (eval a) (eval b)
| Op (Div, a, b) -> eval_number_op ( / ) ( /. ) (eval a) (eval b)
| Call ("sin", e) -> VFloat (Stdlib.sin (as_float (eval e)))
| Call _ -> failwith "unknown function"
```

Now, run the tests using `dune runtest`. Accept the correction with `dune promote`.

## Conclusion

We've added several features to our calculator, and added tests in the meantime. To do so, we've used `dune runtest` and `dune promote`, two of the most useful Dune commands.

## Checkpoint

This is how the project looks like at the end of this chapter.

Listing 11: `dune-project` (unchanged)

```
(lang dune 3.18)
(using menhir 3.0)
(package (name calc))
```

Listing 12: `bin/dune` (unchanged)

```
(executable
 (public_name calc)
 (libraries calc))
```

Listing 13: `bin/calc.ml` (unchanged)

```
let () = Calc.Cli.main ()
```

Listing 14: `lib/dune` (unchanged)

```
(library
 (name calc)
 (libraries cmdliner))

(ocamllex lexer)

(menhir
 (modules parser))
```

Listing 15: lib/ast.ml

```

type op =
| Add
| Mul
| Div

type exp =
| Int of int
| Float of float
| Ident of string
| Op of op * exp * exp
| Call of string * exp

```

Listing 16: lib/cli.ml

```

type value = VInt of int | VFloat of float

let value_to_string = function
| VInt n -> string_of_int n
| VFloat f -> Printf.sprintf "%.6g" f

let eval_number_op f_int f_float va vb =
  match (va, vb) with
  | VInt na, VInt nb -> VInt (f_int na nb)
  | VFloat fa, VFloat fb -> VFloat (f_float fa fb)
  | VInt na, VFloat fb -> VFloat (f_float (float_of_int na) fb)
  | VFloat fa, VInt nb -> VFloat (f_float fa (float_of_int nb))

let as_float = function
| VInt n -> float_of_int n
| VFloat f -> f

let rec eval = function
| Ast.Int n -> VInt n
| Float f -> VFloat f
| Ident "pi" -> VFloat (2. *. Stdlib.acos 0.)
| Ident _ -> failwith "unknown ident"
| Op (Add, a, b) -> eval_number_op ( + ) ( +. ) (eval a) (eval b)
| Op (Mul, a, b) -> eval_number_op ( * ) ( *. ) (eval a) (eval b)
| Op (Div, a, b) -> eval_number_op ( / ) ( /. ) (eval a) (eval b)
| Call ("sin", e) -> VFloat (Stdlib.sin (as_float (eval e)))
| Call _ -> failwith "unknown function"

let info = Cmdliner.Cmd.info "calc"

let eval_lb lb =
  try
    let expr = Parser.main Lexer.token lb in
    let v = eval expr in
    Printf.printf "%s\n" (value_to_string v)
  with Parser.Error ->

```

(continues on next page)

(continued from previous page)

```

Printf.printf "parse error near character %d" lb.lex_curr_pos

let repl () =
  while true do
    Printf.printf ">> %!";
    let lb = Lexing.from_channel Stdlib.stdin in
    eval_lb lb
  done

let term =
  let open Cmdliner.Term.Syntax in
  let+ expr_opt =
    let open Cmdliner.Arg in
    value & opt (some string) None & info [ "e" ]
  in
  match expr_opt with
  | Some s -> eval_lb (Lexing.from_string s)
  | None -> repl ()

let cmd = Cmdliner.Cmd.v info term
let main () = Cmdliner.Cmd.eval cmd |> Stdlib.exit

```

Listing 17: lib/lexer.mll

```

let space = [' ']+
let digit = ['0'-'9']
let letter = ['a'-'z']
let ident = letter+

rule token = parse
  | eof { Parser.Eof }
  | space { token lexbuf }
  | '\n' { Parser.Eof }
  | '+' { Parser.Plus }
  | '*' { Parser.Star }
  | '/' { Parser.Slash }
  | '(' { Parser.Lpar }
  | ')' { Parser.Rpar }
  | digit+ { Parser.Int (int_of_string (Lexing.lexeme lexbuf)) }
  | digit+ '.' digit+ { Parser.Float (float_of_string (Lexing.lexeme lexbuf)) }
  | ident { Parser.Ident (Lexing.lexeme lexbuf) }

```

Listing 18: lib/parser.mly

```

%token Eof
%token<int> Int
%token Plus
%token Star
%token Slash

```

(continues on next page)

(continued from previous page)

```
%token Lpar Rpar
%token<float> Float
%token<string> Ident
%start<Ast.exp> main

%left Plus
%left Star Slash

%{ open Ast %}

%%

main: expr Eof { $1 }

expr:
| Int { Int $1 }
| expr Plus expr { Op (Add, $1, $3) }
| expr Star expr { Op (Mul, $1, $3) }
| expr Slash expr { Op (Div, $1, $3) }
| Ident Lpar expr Rpar { Call ($1, $3) }
| Float { Float $1 }
| Ident { Ident $1 }

%%
```

Listing 19: test/dune (unchanged)

```
(cram
 (deps %bin:calc))
```

Listing 20: test/calc.t

```
$ calc -e '1+2'
3

$ calc -e '1+'
parse error near character 2

$ calc -e '1+2.5'
3.5

$ calc -e '1+pi'
4.14159

$ calc -e '1+2*3'
7

$ calc -e '4/2'
2

$ calc -e 'sin (pi / 6)'
0.5
```

## 2.1.4 Interfacing with C

In this chapter, we're going to extend our calculator with a new function. The difference with `sin` is that we're going to use a C implementation of the function because it is not available in `Stdlib`. To do so, we're going to use (*foreign\_stubs*) to implement the function in C.

### Create a test

Add a new test in `test/calc.t`:

```
$ calc -e 'log10(123456)'
```

Run `dune runtest` and see the failure. Run `dune promote` to add the failure to the test file. Our goal in the rest of the chapter is to change the output of this test.

### Lexing

We have a tiny change to make in `lib/lexer.mll`: extend function names so that they can contain numbers (but not at the beginning).

```
let ident = letter (letter | digit)+
```

### Evaluation

Let's extend our `eval` function in `lib/cli.ml` to handle a `log10` function. Instead of implementing the function in OCaml, we declare it as an external (with its type).

```
external log10_c : float -> float = "calc_log10"

let rec eval = function
| Ast.Int n -> VInt n
| Float f -> VFloat f
| Ident "pi" -> VFloat (2. *. Stdlib.acos 0.)
| Ident _ -> failwith "unknown ident"
| Op (Add, a, b) -> eval_number_op ( + ) ( +. ) (eval a) (eval b)
| Op (Mul, a, b) -> eval_number_op ( * ) ( *. ) (eval a) (eval b)
| Op (Div, a, b) -> eval_number_op ( / ) ( /. ) (eval a) (eval b)
| Call ("sin", e) -> VFloat (Stdlib.sin (as_float (eval e)))
| Call ("log10", e) -> VFloat (log10_c (as_float (eval e)))
| Call _ -> failwith "unknown function"
```

### Create Foreign Stubs

The final part is to implement `calc_log10` as a C function and link it with our library.

Let's create a file `lib/calc_stubs.c`:

```
#include <caml/alloc.h>
#include <caml/memory.h>
#include <math.h>

value calc_log10 (value vx)
{
    CAMLparam1(vx);
    double x = Double_val(vx);
```

(continues on next page)

(continued from previous page)

```

double r = log10(x);
CAMLreturn(caml_copy_double(r));
}

```

**Note**

The interface between C and OCaml code uses a C type called `value`.

Values of this type can be converted from and to `double` using `Double_val` and `caml_copy_double`.

They need to be registered with the garbage collector using the `CAMLparam1` and `CAMLreturn` macros.

And we finally we specify to Dune that this file is part of the library in `lib/dune`:

```

(library
  (name calc)
  (libraries cmdliner)
  (foreign_stubs
    (language c)
    (names calc_stubs)))

```

Run the tests again with `dune runtest`. At that point, the output should be correct. Call `dune promote` to update the expected output.

**Conclusion**

In this chapter, we've extended our library with some C code. The mechanism to do so is called *foreign stubs*.

**Checkpoint**

This is how the project looks like at the end of this chapter.

Listing 21: dune-project (unchanged)

```

(lang dune 3.18)
(using menhir 3.0)
(package (name calc))

```

Listing 22: bin/dune (unchanged)

```

(executable
  (public_name calc)
  (libraries calc))

```

Listing 23: bin/calc.ml (unchanged)

```

let () = Calc.Cli.main ()

```

Listing 24: lib/dune

```

(library
  (name calc)

```

(continues on next page)

(continued from previous page)

```
(libraries cmdliner)
(foreign_stubs
 (language c)
 (names calc_stubs)))

(ocamllex lexer)

(menhir
 (modules parser))
```

Listing 25: lib/ast.ml (unchanged)

```
type op =
| Add
| Mul
| Div

type exp =
| Int of int
| Float of float
| Ident of string
| Op of op * exp * exp
| Call of string * exp
```

Listing 26: lib/calc\_stubs.c

```
#include <caml/alloc.h>
#include <caml/memory.h>
#include <math.h>

value calc_log10 (value vx)
{
    CAMLparam1(vx);
    double x = Double_val(vx);
    double r = log10(x);
    CAMLreturn(caml_copy_double(r));
}
```

Listing 27: lib/cli.ml

```
type value = VInt of int | VFloat of float

let value_to_string = function
| VInt n -> string_of_int n
| VFloat f -> Printf.sprintf "%.6g" f

let eval_number_op f_int f_float va vb =
  match (va, vb) with
  | VInt na, VInt nb -> VInt (f_int na nb)
  | VFloat fa, VFloat fb -> VFloat (f_float fa fb)
  | VInt na, VFloat fb -> VFloat (f_float (float_of_int na) fb)
  | VFloat fa, VInt nb -> VFloat (f_float fa (float_of_int nb))
```

(continues on next page)

```
let as_float = function
| VInt n -> float_of_int n
| VFloat f -> f

external log10_c : float -> float = "calc_log10"

let rec eval = function
| Ast.Int n -> VInt n
| Float f -> VFloat f
| Ident "pi" -> VFloat (2. *. Stdlib.acos 0.)
| Ident _ -> failwith "unknown ident"
| Op (Add, a, b) -> eval_number_op ( + ) ( +. ) (eval a) (eval b)
| Op (Mul, a, b) -> eval_number_op ( * ) ( *. ) (eval a) (eval b)
| Op (Div, a, b) -> eval_number_op ( / ) ( /. ) (eval a) (eval b)
| Call ("sin", e) -> VFloat (Stdlib.sin (as_float (eval e)))
| Call ("log10", e) -> VFloat (log10_c (as_float (eval e)))
| Call _ -> failwith "unknown function"

let info = Cmdliner.Cmd.info "calc"

let eval_lb lb =
  try
    let expr = Parser.main Lexer.token lb in
    let v = eval expr in
    Printf.printf "%s\n" (value_to_string v)
  with Parser.Error ->
    Printf.printf "parse error near character %d" lb.lex_curr_pos

let repl () =
  while true do
    Printf.printf ">> %!";
    let lb = Lexing.from_channel Stdlib.stdin in
    eval_lb lb
  done

let term =
  let open Cmdliner.Term.Syntax in
  let+ expr_opt =
    let open Cmdliner.Arg in
    value & opt (some string) None & info [ "e" ]
  in
  match expr_opt with
  | Some s -> eval_lb (Lexing.from_string s)
  | None -> repl ()

let cmd = Cmdliner.Cmd.v info term
let main () = Cmdliner.Cmd.eval cmd |> Stdlib.exit
```

Listing 28: lib/lexer.mll

```

let space = [' ']+

let digit = ['0'-'9']+

let letter = ['a'-'z']

let ident = letter (letter | digit)+

rule token = parse
  | eof { Parser.Eof }
  | space { token lexbuf }
  | '\n' { Parser.Eof }
  | '+' { Parser.Plus }
  | '*' { Parser.Star }
  | '/' { Parser.Slash }
  | '(' { Parser.Lpar }
  | ')' { Parser.Rpar }
  | digit+ { Parser.Int (int_of_string (Lexing.lexeme lexbuf)) }
  | digit+ '.' digit+ { Parser.Float (float_of_string (Lexing.lexeme lexbuf)) }
  | ident { Parser.Ident (Lexing.lexeme lexbuf) }

```

Listing 29: lib/parser.mly (unchanged)

```

%token Eof
%token<int> Int
%token Plus
%token Star
%token Slash
%token Lpar Rpar
%token<float> Float
%token<string> Ident
%start<Ast.exp> main

%left Plus
%left Star Slash

%{ open Ast %}

%%

main: expr Eof { $1 }

expr:
| Int { Int $1 }
| expr Plus expr { Op (Add, $1, $3) }
| expr Star expr { Op (Mul, $1, $3) }
| expr Slash expr { Op (Div, $1, $3) }
| Ident Lpar expr Rpar { Call ($1, $3) }
| Float { Float $1 }
| Ident { Ident $1 }

```

(continues on next page)

(continued from previous page)

%%

Listing 30: test/dune (unchanged)

```
(cram
 (deps {%bin:calc}))
```

Listing 31: test/calc.t

```
$ calc -e '1+2'
3

$ calc -e '1+'
parse error near character 2

$ calc -e '1+2.5'
3.5

$ calc -e '1+pi'
4.14159

$ calc -e '1+2*3'
7

$ calc -e '4/2'
2

$ calc -e 'sin (pi / 6)'
0.5

$ calc -e 'log10(123456)'
5.09151
```

## 2.1.5 Using a PPX Preprocessor

Our calculator is pretty much opaque: we feed it a string, and it displays a result (on an error message), but we have now way to know what the internal expression looks like.

In this chapter, we're going to use a ppx deriver to generate a `pp_expr` function that can display expressions.

### Prerequisites

Install `ppx_deriving` by running:

```
opam install ppx_deriving.5.2.1
```

### Create a test

Add a new test in `test/calc.t`:

```
$ calc --debug-ast -e '2 * sin (pi / 2)'
```

Run `dune runtest` and see the failure. Run `dune promote` to add the failure to the test file. Our goal in the rest of the chapter is to change the output of this test.

### Use `ppx_deriving.show`

Add an `[@@deriving show]` attribute on types in `lib/ast.ml`:

```
type op =
| Add
| Mul
| Div
[@@deriving show]

type expr =
| Int of int
| Float of float
| Ident of string
| Op of op * expr * expr
| Call of string * expr
[@@deriving show]
```

This will generate `pp_op`, `show_op`, `pp_expr`, and `show_expr` functions.

To do so we need to instruct Dune to use `ppx_deriving.show` as a preprocessor by updating `lib/dune`:

```
(library
 (name calc)
 (libraries cmdliner)
 (preprocess
  (pps ppx_deriving.show))
 (foreign_stubs
  (language c)
  (names calc_stubs)))
```

### Add a `--debug-ast` Flag

We have a few edits to make to `lib/cli.ml`.

First, add a new `cmdliner` flag to parse the command-line and pass it to `repl` and `eval_lb`:

```
let term =
  let open Cmdliner.Term.Syntax in
  let+ expr_opt =
    let open Cmdliner.Arg in
    value & opt (some string) None & info [ "e" ]
  and+ debug_ast =
    let open Cmdliner.Arg in
    value & flag & info [ "debug-ast" ]
  in
  match expr_opt with
  | Some s -> eval_lb ~debug_ast (Lexing.from_string s)
  | None -> repl ~debug_ast
```

Then, forward it from `repl` to `eval_lb`:

```
let repl ~debug_ast =
  while true do
    Printf.printf ">> %!";
    let lb = Lexing.from_channel Stdlib.stdin in
    eval_lb ~debug_ast lb
  done
```

Finally, update `eval_lb` to use it:

```
let eval_lb ~debug_ast lb =
  try
    let expr = Parser.main Lexer.token lb in
    if debug_ast then Format.eprintf "[debug] %a\n" Ast.pp_exp expr;
    let v = eval expr in
    Printf.printf "%s\n" (value_to_string v)
  with Parser.Error ->
    Printf.printf "parse error near character %d" lb.lex_curr_pos
```

Run the tests again with `dune runtest`. At that point, the output should be correct. Call `dune promote` to update the expected output.

### Checkpoint

This is how the project looks like at the end of this chapter.

Listing 32: `dune-project` (unchanged)

```
(lang dune 3.18)
(using menhir 3.0)
(package (name calc))
```

Listing 33: `bin/dune` (unchanged)

```
(executable
 (public_name calc)
 (libraries calc))
```

Listing 34: `bin/calc.ml` (unchanged)

```
let () = Calc.Cli.main ()
```

Listing 35: `lib/dune`

```
(library
 (name calc)
 (libraries cmdliner)
 (foreign_stubs
 (language c)
 (names calc_stubs))
 (preprocess
 (pps ppx_deriving.show)))

(ocamllex lexer)
```

(continues on next page)

(continued from previous page)

```
(menhir
(modules parser))
```

Listing 36: lib/ast.ml

```
type op =
| Add
| Mul
| Div
[@@deriving show]

type exp =
| Int of int
| Float of float
| Ident of string
| Op of op * exp * exp
| Call of string * exp
[@@deriving show]
```

Listing 37: lib/calc\_stubs.c (unchanged)

```
#include <caml/alloc.h>
#include <caml/memory.h>
#include <math.h>

value calc_log10 (value vx)
{
  CAMLparam1(vx);
  double x = Double_val(vx);
  double r = log10(x);
  CAMLreturn(caml_copy_double(r));
}
```

Listing 38: lib/cli.ml

```
type value = VInt of int | VFloat of float

let value_to_string = function
| VInt n -> string_of_int n
| VFloat f -> Printf.sprintf "%.6g" f

let eval_number_op f_int f_float va vb =
  match (va, vb) with
  | VInt na, VInt nb -> VInt (f_int na nb)
  | VFloat fa, VFloat fb -> VFloat (f_float fa fb)
  | VInt na, VFloat fb -> VFloat (f_float (float_of_int na) fb)
  | VFloat fa, VInt nb -> VFloat (f_float fa (float_of_int nb))

let as_float = function
| VInt n -> float_of_int n
| VFloat f -> f
```

(continues on next page)

```

external log10_c : float -> float = "calc_log10"

let rec eval = function
| Ast.Int n -> VInt n
| Float f -> VFloat f
| Ident "pi" -> VFloat (2. *. Stdlib.acos 0.)
| Ident _ -> failwith "unknown ident"
| Op (Add, a, b) -> eval_number_op ( + ) ( +. ) (eval a) (eval b)
| Op (Mul, a, b) -> eval_number_op ( * ) ( *. ) (eval a) (eval b)
| Op (Div, a, b) -> eval_number_op ( / ) ( /. ) (eval a) (eval b)
| Call ("sin", e) -> VFloat (Stdlib.sin (as_float (eval e)))
| Call ("log10", e) -> VFloat (log10_c (as_float (eval e)))
| Call _ -> failwith "unknown function"

let info = Cmdliner.Cmd.info "calc"

let eval_lb ~debug_ast lb =
  try
    let expr = Parser.main Lexer.token lb in
    if debug_ast then Format.eprintf "[debug] %a\n" Ast.pp_exp expr;
    let v = eval expr in
    Printf.printf "%s\n" (value_to_string v)
  with Parser.Error ->
    Printf.printf "parse error near character %d" lb.lex_curr_pos

let repl ~debug_ast =
  while true do
    Printf.printf ">> %!";
    let lb = Lexing.from_channel Stdlib.stdin in
    eval_lb ~debug_ast lb
  done

let term =
  let open Cmdliner.Term.Syntax in
  let+ expr_opt =
    let open Cmdliner.Arg in
    value & opt (some string) None & info [ "e" ]
  and+ debug_ast =
    let open Cmdliner.Arg in
    value & flag & info [ "debug-ast" ]
  in
  match expr_opt with
  | Some s -> eval_lb ~debug_ast (Lexing.from_string s)
  | None -> repl ~debug_ast

let cmd = Cmdliner.Cmd.v info term
let main () = Cmdliner.Cmd.eval cmd |> Stdlib.exit

```

Listing 39: lib/lexer.mll

```

let space = [' ']+

let digit = ['0'-'9']+

let letter = ['a'-'z']

let ident = letter (letter | digit)+

rule token = parse
  | eof { Parser.Eof }
  | space { token lexbuf }
  | '\n' { Parser.Eof }
  | '+' { Parser.Plus }
  | '*' { Parser.Star }
  | '/' { Parser.Slash }
  | '(' { Parser.Lpar }
  | ')' { Parser.Rpar }
  | digit+ { Parser.Int (int_of_string (Lexing.lexeme lexbuf)) }
  | digit+ '.' digit+ { Parser.Float (float_of_string (Lexing.lexeme lexbuf)) }
  | ident { Parser.Ident (Lexing.lexeme lexbuf) }

```

Listing 40: lib/parser.mly (unchanged)

```

%token Eof
%token<int> Int
%token Plus
%token Star
%token Slash
%token Lpar Rpar
%token<float> Float
%token<string> Ident
%start<Ast.exp> main

%left Plus
%left Star Slash

%{ open Ast %}

%%

main: expr Eof { $1 }

expr:
| Int { Int $1 }
| expr Plus expr { Op (Add, $1, $3) }
| expr Star expr { Op (Mul, $1, $3) }
| expr Slash expr { Op (Div, $1, $3) }
| Ident Lpar expr Rpar { Call ($1, $3) }
| Float { Float $1 }
| Ident { Ident $1 }

```

(continues on next page)

(continued from previous page)

%%

Listing 41: test/dune (unchanged)

```
(cram
 (deps %{{bin:calc}}))
```

Listing 42: test/calc.t

```
$ calc -e '1+2'
3

$ calc -e '1+'
parse error near character 2

$ calc -e '1+2.5'
3.5

$ calc -e '1+pi'
4.14159

$ calc -e '1+2*3'
7

$ calc -e '4/2'
2

$ calc -e 'sin (pi / 6)'
0.5

$ calc -e 'log10(123456)'
5.09151

$ calc --debug-ast -e '2 * sin (pi / 2)'
2
[debug] (Ast.Op (Ast.Mul, (Ast.Int 2),
                  (Ast.Call ("sin",
                            (Ast.Op (Ast.Div, (Ast.Ident "pi"), (Ast.Int 2))))))
        ))
```

## 2.1.6 Unit Tests

We're testing our calculator using cram tests, but in some cases, it can be more comfortable to use unit tests to test internal behaviors.

In this chapter, we're going to extract the function that deals with conversion to floats and put it in a test harness.

## Install Dependencies

Run `opam install alcotest.1.7.0`.

## Refactor

Let's refactor our `eval_number_op` function in `lib/cli.ml` so that it uses the `as_float` function. Move `as_float` above it so that it can be used.

```
let eval_number_op f_int f_float va vb =
  match (va, vb) with
  | VInt na, VInt nb -> VInt (f_int na nb)
  | _ ->
    let fa = as_float va in
    let fb = as_float vb in
    VFloat (f_float fa fb)
```

## Create a Test Suite

We're first going to move our cram tests so that they can be in `test/cram/`, while our unit tests will be in `test/unit/`.

Move the contents of the folder `test` into a fresh folder `test/cram/`.

Create a folder `test/unit/`.

Create `test/unit/dune` with the following contents:

```
(test
 (name test_calc)
 (libraries alcotest calc))
```

And `test/unit/test_calc.ml`:

```
open Calc

let test_as_float =
  let test ~name expression ~expected =
    ( Printf.sprintf "as_float(%s)" name,
      `Quick,
      fun () ->
        let got = Cli.as_float expression in
        Alcotest.check
          (Alcotest.float Stdlib.epsilon_float)
          __LOC__ expected got )
  in
  [
    test ~name:"int" (VInt 2) ~expected:2.;
    test ~name:"float" (VFloat 3.5) ~expected:3.5;
  ]

let suite = [ ("as_float", test_as_float) ]

let () = Alcotest.run __FILE__ suite
```

Now run `dune runtest`. In addition to the existing cram tests, this also runs unit tests.

### 2.1.7 Conclusion

This tutorial has covered the parts of Dune that users are the most likely to interact with:

- the *executable*, *library*, and *test* stanzas;
- how to use *cram tests* and the workflow of *promotion*;
- bindings to C code using *foreign stubs*;
- using a ppx deriver.

#### Where to go from here

You might be interested in:

- our *how-to guides* to apply this to your projects;
- various *explanations* about how Dune works.

## 2.2 OCaml Package Management With Dune

### Warning

Dune Package Management is not final yet and details are still subject to change.

In this tutorial we will be looking at how to use Dune for managing project dependencies. This enables users to install the compiler as well as third-party dependencies using a single tool which takes care of building code and dependencies.

To get started you only need Dune. Head to *Setting Up Package Management With Dune* to begin the setup.

### 2.2.1 Setting Up Package Management With Dune

The idea of package management with Dune has been as unobtrusive as possible. Thus most projects can easily be built with just the minimum of changes.

In this tutorial we will create a simple project to use the integrated package management feature for the very first time.

#### Declare Dependencies

The best way to work with the package management is to declare your dependencies in the `dune-project` file.

##### `dune-project`

```
(lang dune 3.17)
(name test)

(package
  (name test)
  (depends
    (ocaml (>= 4.14))))
```

We define a project called `test` and declare that to build it we need an OCaml compiler that is at least version 4.14.

This is exactly the same information that is used to generate opam files using the `generate_opam_files` stanza as described in *How to Generate Opam Files from dune-project*.

`test.ml`

```
let langs = ["OCaml"; "Rust"]

let () =
  let s = String.concat ", " langs in
  Format.printf "Hello, %s!\n" s
```

To show that the build works, this simple program will be built and executed.

`dune`

```
(executable
 (public_name test))
```

To declare our module an executable we need a little bit of configuration, so we just define the module as an executable. After our project skeleton is set up, we can proceed to the next step.

### Enable package management

So far we have done everything as in a regular Dune project. However, now we need to tell Dune that we want to use the package management feature. To do so we have to add a new stanza to our `dune-workspace` file, creating it if it doesn't exist.

`dune-workspace`

```
(lang dune 3.21)
(pkg enabled)
```

This new stanza will tell Dune to determine the packages that your project depends on, find the right versions, download, and build them on the next build.

### Build Project

To build the project, you can just use the regular Dune commands.

```
dune build
```

Since this is the first time we have run the build system after enabling package management a number of things will happen:

1. It will download and update all required opam repositories to determine which packages are available.
2. It will attempt to find a package solution that satisfies all dependency constraints.
3. It will download the sources of the dependencies.
4. It will build the dependencies in sandbox locations.
5. It will install the dependencies in the build folder.
6. It will build the project using the dependencies that it has installed.

This means that building the first time will take longer than usual, as the dependencies need to be built, possibly including the OCaml compiler. Subsequent builds where all dependencies have already been built will be significantly faster.

We can show that the package has been built successfully and works as expected:

```
$ dune exec ./test.exe
Hello, OCaml, Rust!
```

### Note

If you don't want to build your project, instead stopping at step 5, you can use the `@pkg-install` alias like so

```
$ dune build @pkg-install
```

This functionality can be useful to cache the installation of dependencies, somewhat similar to `opam switch create` followed by `opam install`.

See *@pkg-install* for more information.

## Conclusion

In this section we learned how to set up a Dune project that picks a compiler and installs it without the need for any additional tooling.

In the next section *Managing Dependencies* we will look at how to add third party dependencies.

## 2.2.2 Managing Dependencies

The OCaml ecosystem has a wealth of third-party packages that are available for use. In this section we will look into how to use them with Dune.

### Adding Dependencies

Much like in regular projects, to add a library we need to add a dependency to it. For simplicity we will use the popular `fmt` library as an example, but any package from the [package repository](#) can be used.

To do so we update the `dune-project` file to add a dependency on the `opam` package.

#### `dune-project`

```
(lang dune 3.17)
(name test)

(package
  (name test)
  (depends
    (ocaml (>= 4.14))
    fmt))
```

Here we define the OPAM packages that we want to use, along with the version constraints these dependencies should adhere to.

#### `dune-workspace`

```
(lang dune 3.21)
(pkg enabled)
```

In this file we direct Dune to enable package management in the current workspace. The `pkg` stanza configures Dune to manage the declared dependencies automatically.

This configuration will take care of installing the dependencies, but we still need to add it to our build as a library as usual:

### dune

```
(executable
  (public_name test)
  (libraries fmt))
```

Adding a library dependency to our dune file via the `libraries` stanza. This is unchanged from the usual Dune workflow.

This change will allow us to use the `Fmt` module in our OCaml code.

### test.ml

```
let langs = ["OCaml"; "Rust"]

let () =
  let pp_langs = Fmt.(list ~sep:(any ", ") string) in
  Format.printf "Hello, %a!\n" pp_langs langs
```

We update the code to define an `Fmt.t` pretty-printer for the list of strings and then use it to print the value.

To build it we just call `build` again.

```
$ dune build
```

Dune will notice that the project depends on new packages. Thus it will re-run the internal dependency solver to find a solution for the set of packages to use. It will then use this new solution to download, build and install these dependencies automatically.

#### Note

The installed packages will includes all the dependencies required by your project, including transitive dependencies.

As we see, the code works and uses `fmt` to do the pretty-printing:

```
$ dune exec ./test.exe
Hello, OCaml, Rust!
```

## Dependency Constraints

Packages are often only compatible with some versions of dependencies. To specify a version range, use the regular Dune dependency syntax used for `opam` dependencies in the `dune-project` file.

### dune-project

```
(lang dune 3.17)
(name test)

(package
```

(continues on next page)

(continued from previous page)

```
(name test)
(depends
 (ocaml (>= 4.14))
 (fmt (and (>= 0.6) (< 1.0))))))
```

This change ensures the `fmt` package to install will be compatible with our request. These constraints will be taken into account the next time the build system is run.

```
dune build
```

### Removing Dependencies

Given all dependencies are defined in the `dune-project` file, removing a dependency just means to remove the dependency from the `depends` field of your `dune-project`.

From then on the project will not depend on the package anymore, and in future builds the package will not be accessible as `library` anymore.

#### Note

The removed dependency might still be accessible if some other dependency of your project depends on it, thus if it is a transitive dependency.

### External Dependencies

Many packages also declare external system dependencies (“depexts”) that must be installed in order to compile or use the package. This information is stored in the package’s `.opam` file.

To view these external dependencies, use `dune show depexts`. The output can then be used to install the required system packages via your system package manager.

For example, the `postgresql` library shows the following depexts on an Ubuntu system:

```
$ dune show depexts
libpq-dev
pkg-config
```

### 2.2.3 Pinning Projects

When Dune is looking up packages to lock, it uses the (pre)configured OCaml package repositories. However it is also possible to manually specify the sources of packages; for example, if the package is not released in a package repository. This is called “pinning”.

#### Installing Packages From a Pin

To pin a package, a new `pin` has to be declared in the `dune-project` file.

```
dune-project
```

```
(lang dune 3.17)
(name test)

(pin
```

(continues on next page)

(continued from previous page)

```
(url "git+https://github.com/dbuenzli/fmt.git")
(package (name fmt)))

(package
 (name test)
 (depends
  (ocaml (>= 4.14))
  fmt))
```

This will create a pin on the `fmt` package and use the specified Git repository URL to retrieve the sources. For more information refer to *the pin stanza reference*.

Don't forget to remove the version constraints from `fmt` in the list of dependencies.

The next time the project is built, Dune will use this repository instead of the information from the selected package repositories. Thus the `fmt` package will be built from the source in the specified Git repository rather than from the source tarball released in the `opam-repository`.

```
dune build
```

Unlike previously, the version of the `fmt` library that is picked is `dev`, to signify a development version.

```
$ dune exec ./test.exe
Hello, OCaml, Rust!
```

## 2.2.4 Custom Repositories

By default when locking package versions, Dune looks up packages from two sources:

1. The upstream, community maintained `opam-repository` at `ocaml/opam-repository` for most packages
2. An overlay repository with patched software to allow usage in Dune at `ocaml-dune/opam-overlays`

To change the presets, the `dune-workspace` file has to be edited (and created if it didn't exist):

### `dune-workspace`

```
(lang dune 3.21)

(pkg enabled)

(lock_dir
 (repositories overlay specific-upstream))

(repository
 (name specific-upstream)
 (url "git+https://github.com/ocaml/opam-repository.git
 ↪️#00ac3727bc4ac0eabd3c89e69c1660d6b63a3d48"))
```

In this case, we want to select a specific revision of the community repository instead of always using the most recent one as it would do by default. We define a new repository and configure the Dune solver to use this repository.

For more information about the stanzas refer to *the repositories stanza* as well as *the lock\_dir stanza*.

The next time the build system is run, instead of using the default repositories at their newest versions, the solver will check out the configured repositories at the defined revisions. These will then be used to determine the new solution, which will get used for downloading and building the dependencies.

### Note

This feature can also be used to make sure the locked dependencies are reproducible, as fixing all the package repository versions will lead to equivalent locking results.

### See also

*Locking Your Dependencies* More information how to keep package versions locked.

## 2.2.5 Locking Your Dependencies

In the default use-case Dune will automatically determine which packages to install, by reading the package constraints, determining compatible versions and installing the dependencies automatically.

For many projects this is a good and acceptable behavior as users often want to use new versions of their dependencies. However some projects might want to keep a fixed set of (transitive) dependencies that is only updated manually.

### Creating a lock directory

If a lock directory exists in the source, Dune will use that to fix the exact version and source of dependencies. The default name of said lock directory is `dune.lock`.

Lock directories are created with:

```
$ dune pkg lock
Solution for dune.lock:
- ocaml.5.2.0
- ocaml-base-compiler.5.2.0
- ocaml-config.3
```

Whenever Dune encounters a `dune.lock` folder, it will use the set of dependencies defined in the lock. It contains all the metadata necessary to build a project's dependencies, including every packages' name, version, dependencies, source location.

On the next build, Dune will read the stored solver solution from the `dune.lock` directory, download and build the dependencies, and then continue to build the project as usual.

### Note

This approach is similar to using `opam switch export --full --freeze` to export the configuration of a switch.

### Updating a lock directory

To update a lock directory, rerun

```
$ dune pkg lock
```

All the dependencies in the lock directory will be updated to the latest available versions that remain consistent with the declared constraints.

## Removing a lock directory

Deleting the lock directory will cause Dune to fall back to automatically determining dependency versions via the declared package constraints.

## 2.2.6 Setting up an OCaml project

As an example of how to add custom repositories and pin dependencies in projects using Dune Package Management, let's look at setting up a basic [OCaml](#) project. OCaml is a compiler branch with a fast moving set of extensions to OCaml. Some tweaks are required to set up a project that compiles with OCaml when using Dune package management.

### Setting up the dune-workspace

OCaml has a custom `opam-repository` that provides compatible dependencies. Dune can use packages from these repositories by configuring it in the `dune-workspace` file using the *repositories stanza* and *lock\_dir stanza*.

```
(lang dune 3.21)

(pkg enabled)

(repository
 (name oxcaml)
 (url git+https://github.com/oxcaml/opam-repository))

(lock_dir
 (repositories overlay oxcaml upstream))
```

We would like dune to pick the OCaml compiler rather than the latest upstream one. This can be achieved by manually adding a constraint on the compiler that is picked in the *package stanza* in the `dune-project`.

```
(lang dune 3.21)

(package
 (name hello-oxcaml)
 (depends
 (ocaml-variants
 (= 5.2.0+ox))))
```

Let's add a test program that uses OCaml syntax.

#### main.ml

```
let () =
  (* The `local_` keyword requires OCaml *)
  let local_ i = 42 in
  let j = i + 1 in
  Printf.printf "%d\n" j
```

And to build it, we add a dune file.

```
(executable
 (public_name main))
```

### Building the project

Let's compile and execute the project. Note that when you run `dune pkg lock` or `dune build` the very first time, Dune pulls in and builds all the dependencies before building the project itself. It is expected to take a considerable amount of time.

Note: With autolocking enabled one doesn't need to explicitly run `dune pkg lock` to generate a lock directory. However, a known issue, [dune#12851](#) currently prevents us from using autolocking.

```
$ dune pkg lock
Solution for dune.lock
```

Dependencies common to all supported platforms:

- `conf-autoconf.0.2`
- `conf-which.1`
- `ocaml.5.2.0`
- `ocaml-config.3`
- `ocaml-variants.5.2.0+ox`

```
$ dune exec ./main.exe
...
43
```

We have successfully built the project with the OCaml compiler and run the executable.

### Developer tools

Note that OCaml provides its own forks of the developer tools `ocaml-lsp-server`, `ocamlformat`, etc. Make sure to include constraints for the tools in the `dune-workspace` if you intend to use them for your development workflows.

Please see [How to Customize Lock Directories of Developer Tools](#) for details about configuring your workspace for developer tools. Please be aware that there are known issues in the current Dune release that may prevent this functionality from working reliably.

## 2.3 OCaml Parameterised Libraries with Dune

### Warning

- This feature is experimental.
- Parameterised libraries are only supported by the OCaml branch of the OCaml compiler. You need to [install the OCaml compiler](#) to use parameterised libraries with Dune.

This tutorial explains how to create and use a parameterized library in Dune. *Parameterised libraries* specify some of their dependencies as *library parameters*, without committing a concrete implementations. This is useful for build-time dependency injection and for generalizing over platform-specific implementations. It provides a more flexible alternative to dune's *Virtual Libraries*.

By the end of the tutorial, you will know:

- how to declare a library parameter,
- how to implement a library parameter,
- how to parameterise a library on a library parameter,
- how to instantiate a parameterised library by supplying an implementation of its library parameters.

### 2.3.1 Getting Started

Once you have checked that [OxCaml](#) is installed and its `ocamlc` is available on your path, you can enable Dune's `ocaml` extension in your `dune-project`:

```
(lang dune 3.20)

(using ocaml 0.1)
```

#### Important

Enabling the OxCaml extension by adding the `(use ocaml 0.1)` stanza allows dune to use the stanzas and fields specific to OxCaml. This extension is only available since dune 3.20.

### 2.3.2 Declare a Library Parameter

Library parameters allow libraries to be generic over libraries they depend on. This is useful for dependency injection, testing, platform-specific code, etc.

#### Note

The interface of a library parameter is specified using an OCaml `.mli` file and declared via the `library_parameter` dune stanza. The following analogy holds:

	module parameters	library parameters
<i>declaration</i>	<code>module type Intf = &lt;sig&gt;</code>	<code>(library_parameter (name intf)) stanza</code>
<i>specification</i>	a module signature: <code>sig ... end</code>	an interface file: e.g., <code>intf.mli</code>

First, we create an (arbitrarily named) `param/` directory where we are going to write our parameter definition.

```
mkdir param
```

We create a `param/dune` file containing a `library_parameter` stanza that declares a new library parameter named `param`:

```
(library_parameter
 (name param))
```

To complete the declaration, we must provide an `.mli` file matching the given name, which will specify the interface of the library parameter. We create this in `param/param.mli`:

```
type t

val make : unit -> t

val print : t -> unit
```

This interface specifies the types, functions, and modules which must be provided by a library for it to implement the declared parameter.

### 2.3.3 Implement a Library Parameter

#### Note

The implementation of a library parameter must satisfy the signature of the parameter it claims to implement. The following analogy holds:

	module implementation	library implementation
<i>ascribed declaration</i>	<code>module Impl : Intf</code>	a library stanza with an <code>implements</code> field: <code>(library (name impl) (implements intf))</code>
<i>implementation</i>	a module expression: <code>struct ... end</code>	a set of <code>.ml</code> files

We create an (arbitrarily named) directory `impl/` where we are going to write an implementation of our library parameter:

```
mkdir impl
```

An implementation of a declared library parameter named `param` is a normal dune library with the `(implements param)` annotation. We declare this `impl/dune`:

```
(library
 (name impl)
 (implements param))
```

For the sake of simplicity, we give the example of a single-module library, defined in `impl/impl.ml`. This must satisfy the parameter interface ascribed by the `implements` field:

```
type t = string

let make () = "impl"

let print t = print_endline t
```

### 2.3.4 Define a Parameterised Library

#### Note

A parameterised library must specify the parameters it requires. The following analogy holds:

	parameterised module (a “functor”)	parameterised library
<i>declaration</i>	<code>module M (P : Intf)</code>	a <code>(library (name m) (parameters intf))</code> stanza

We create an (arbitrarily named) directory `lib/` where we are going to write a library that is parameterised on our declared library parameter.

```
mkdir lib
```

We add a `lib/dune` file to declare the library, with the `(parameters param)` field, indicating that the library is generic over the library parameter `param`:

```
(library
 (name lib)
 (parameters param))
```

Within the modules comprising a parameterised library, we can then refer library parameters without having to decide on a concrete implementation of the libraries up front. In our example, we can directly use the functions from the declared `Param` in our library `lib/lib.ml`:

```
let lib () = Param.print (Param.make ())
```

### 2.3.5 Instantiate a Parameterised Library

#### **Note**

To instantiate a parameterised library, we must supply library arguments that implement all specified parameters. The following analogy holds:

	functor application	library instantiation
<i>instantiation</i>	<code>M (P)</code>	a <code>(instantiate m p)</code> expression in <i>Library Dependencies</i>

A *library* can depend on parameterised Libraries if it is, either, parameterised with the same parameters as its dependencies, or, it **instantiates** its parameterised dependencies. An *executable*, however, must instantiate any parameterised libraries, otherwise we wouldn't be able to compile a complete program to execute.

To conclude this tutorial, we define an executable target in the new folder `bin/`, that depends on an instantiation of our parameterised library.

We create an (arbitrarily named) `bin` directory:

```
mkdir bin
```

Then we add a `bin/dune` file defining our executable:

```
(executable
 (name bin)
 (libraries
 (instantiate lib impl)))
```

In the `libraries` field, the syntax `(instantiate lib impl)` specifies that we want to use the concrete library resulting from applying the parameterised library `lib` to the library argument `impl`. Note that this application is only possible because we defined the `impl` library as an implementation of the library parameter, `param`.

The code for our binary simply calls the library code, using the library's name, `Lib`:

```
let () = Lib.lib ()
```

### 2.3.6 Conclusion

In this tutorial, we have learned how to declare a library parameter, define its implementation, define a parameterised library, and finally to use the latter by instantiating it for an executable target.

## HOW-TO GUIDES

These guides will help you use Dune’s features in your project.

### 3.1 How to Install Dune

Dune is available as an Opam package. First, make sure that Opam is installed:

```
$ opam --version  
2.1.5
```

Any version higher than 2.0.0 is supported, though preferably at least 2.1.0.

If Opam is not available, follow [the official instructions on the Opam website](#) to install it and then run its global setup with `opam init`.

#### Note

Opam requires a “shell hook” to work properly. Make sure to set it up correctly during `opam init`. Otherwise you will have to run `eval $(opam env)` every time you create an Opam switch or change directory.

Then, you can install Dune in an Opam switch using the following command:

```
$ opam install dune
```

After the command completes, the following should display a version number:

```
$ dune --version  
3.12.1
```

#### Note

In most cases, when using Opam you will not need to install Dune by hand. Installing the project’s dependencies will install it in the Opam switch.

### 3.2 How to Set up Automatic Formatting

This guide will show you how to configure Dune so that it can check the formatting of your source code.

Formatting is defined per project. This ensures that if a project is reused elsewhere, its formatting configuration will not interfere.

### 3.2.1 Setting Up the Environment

First, let's open the `dune-project` file. Make sure that the version specified in `(lang dune X.Y)` is at least `2.0`. Most formatting configuration happens in that file. If you want to format OCaml sources and dune files, you don't have anything to add. Otherwise, refer to the *formatting* stanza.

Next we need to install some code formatting tools. For OCaml code, this means installing `OCamlFormat` with `opam install ocamlformat`. Formatting dune files is built into Dune and does not require any extra tools. For Reason code, this uses the `refmt` tool which is already installed if you are using Reason syntax in your project. If your project uses a *dialect*, a specific tool might be required.

Using `OCamlFormat` requires some configuration. Take note of the version returned by `ocamlformat --version` (let's name that `X.Y.Z`) and create an `.ocamlformat` file in the same directory as `dune-project` with the following contents:

```
version=X.Y.Z
profile=default
```

The version line is checked by `OCamlFormat` and ensures that everybody contributing to the project uses the same version.

Note that you do not have to add `ocamlformat` to your `opam` files.

### 3.2.2 Running the Formatters

Run the `dune build @fmt` command. It will format the source files in the corresponding project and display the differences:

```
$ dune build @fmt
--- hello.ml
+++ hello.ml.formatted
@@ -1,3 +1 @@
-let () =
-  print_endline
-  "hello, world"
+let () = print_endline "hello, world"
```

Then it's possible to accept the correction by calling `dune promote` to replace the source files with the corrected versions.

```
$ dune promote
Promoting _build/default/hello.ml.formatted to hello.ml.
```

As usual with promotion, it's possible to combine these two steps by running `dune build @fmt --auto-promote`. This command can also be shortened to `dune fmt`. See *Diffing and Promotion* for more details.

### 3.2.3 Setting Up Your CI

To check formatting in CI, the precise set up depends on the CI system used, but in general it is easier to set up a dedicated job that just installs `dune` and the formatting tools, rather than doing that as part of the jobs that run tests.

If you use `ocaml-ci`, you have nothing to do: a formatting job is set up automatically.

If you use `setup-ocaml`, you can use the `lint-fmt` extend listed in the README file.

## 3.3 How to Generate Opam Files from dune-project

This guide will show you how to configure Dune so that it generates opam files.

### 3.3.1 Declaring Package Dependencies

The goal of this first step is to add (package) stanzas in your `dune-project` file. These stanzas declare the metadata that your package uses in the language of opam packages. See *Declaring a Package*.

The next step depends on whether you are starting from a clean slate (new package) or adapting an existing opam file.

#### For a New Package (No Existing Opam File)

If your project does not have any opam files, you will have to find your package dependencies. In the simple case, collect all the libraries that appear in the (libraries) fields of your project and put this list in the (depends) field of the corresponding (package). See *The OCaml Ecosystem* for the difference between libraries and packages.

Example: you have a library that looks like:

```
(library
 (public_name frobnitz)
 (libraries lwt fmt))
```

You can declare the package as:

```
(package
 (name frobnitz)
 (depends lwt fmt))
```

Also add common metadata using (authors), (maintainers), (license), (source), as well as a (synopsis) and a (description) for

#### For an Existing Package

If you already have an opam file (or several of them), you can convert it by following the rules in *package*.

For example, if your opam file looks like:

```
opam-version: 2.0
authors: ["Anil Madhavapeddy" "Rudi Grinberg"]
maintainer: ["team@mirage.org"]
name: "cohttp-async"
synopsis: "HTTP client and server for the Async library"
description: "A _really_ long description"
license: "ISC"
bug-reports: "https://github.com/mirage/ocaml-cohttp/issues"
homepage: "https://github.com/mirage/ocaml-cohttp/"
dev-repo: "git+https://github.com/mirage/ocaml-cohttp.git"
build: [
  ["dune" "subst"] {dev}
  [
    "dune"
    "build"
    "-p"
    name
    "-j"
```

(continues on next page)

```

jobs
  "@install"
  "@runtest" {with-test}
  "@doc" {with-doc}
]
]
depends: [
  "dune" { >= "3.4" }
  "odoc" { with-doc }
  "cohttp" { >= "1.0.2" }
  "conduit-async" { >= "1.0.3" }
  "async" { >= "v0.10.0" }
]
x-maintenance-intent: [ "(latest)" ]

```

You can express this as:

```

(source (github mirage/ocaml-cohttp))
(license ISC)
(authors "Anil Madhavapeddy" "Rudi Grinberg")
(maintainers "team@mirage.org")
(maintenance_intent "(latest)")

(package
  (name cohttp-async)
  (synopsis "HTTP client and server for the Async library")
  (description "A _really_ long description")
  (depends
    (cohttp (>= 1.0.2))
    (conduit-async (>= 1.0.3))
    (async (>= v0.10.0))))

```

## General Notes and Tips

- Do not declare a dependency on the dune and odoc packages. Dune will generate them with the right constraints.
- For fields that are common between packages (like (authors) or (license)), you can use a global one rather than replicate it between packages.
- If you use a platform such as GitHub you can use (source) as a shorthand instead of specifying (bug\_reports), (homepage), etc.
- (package) stanzas do not support all opam fields or complete syntax for dependency specifications. If the package you are adapting requires this, keep the corresponding opam fields in a pkg.opam.template file. See *Packages*.
- It is not necessary to specify (version), this will be added at release time if you use `dune-release`.
- To generate an opam variable such as version, use a colon : followed by the name of the variable. For example, to generate a { = version } in the opam file, use (a (= :version)) in dune-project.

### 3.3.2 Generating Opam Files

If you have existing `*.opam` files, make a backup of them because the instructions in this section will propose updated versions of them.

Now that you have declared package metadata in `dune-project`, you can add `(generate_opam_files)` in `(dune-project)`.

To regenerate the opam files from the metadata in `(package)` stanzas, build the dedicated alias:

```
$ dune build @opam
```

With `(lang dune 3.23)` or newer, if Dune reports a diff, apply it with `dune promote`.

With older Dune language versions, building `@opam` updates the checked-in `*.opam` files directly.

The `@install` and `@runtest` aliases also trigger opam file generation.

#### ➔ See also

##### ***dep\_specification***

How `(depends)` and similar fields are processed.

##### ***How Dune integrates with opam***

How `with-test` and related variables are used by opam.

## 3.4 Cross-Compilation

Dune allows for cross-compilation by defining build contexts with multiple targets. Targets are specified by adding a `targets` field to the build context definition.

`targets` takes a list of target name. It can be either:

- `native`, the native tools that can build binaries to run on the machine doing the build
- the name of an alternative toolchain

Note that at the moment, there is no official support for cross-compilation in OCaml. Dune supports the *opam-cross-`<x>`* repositories from the [OCaml-cross organization on GitHub](#), such as:

- `opam-cross-windows`
- `opam-cross-android`
- `opam-cross-ios`

In particular:

- to build Windows binaries using `opam-cross-windows`, write `windows` in the list of targets
- to build Android binaries using `opam-cross-android`, write `android` in the list of targets
- to build IOS binaries using `opam-cross-ios`, write `ios` in the list of targets

For example, the following workspace file defines three different targets for the `default` build context:

```
(context (default (targets native windows android)))
```

This configuration defines three build contexts:

- `default`

- `default.windows`
- `default.android`

Note that the `native` target is always implicitly added when not present; however, `dune build @install` will skip this context, i.e., `default` will only be used for building executables needed by the other contexts.

With such a setup, calling `dune build @install` will build all the packages three times.

Note that instead of writing a `dune-workspace` file, you can also use the `-x` command line option. Passing `-x foo` to `dune` without having a `dune-workspace` file is the same as writing the following `dune-workspace` file:

```
(context (default (targets foo)))
```

If you have a `dune-workspace` and pass a `-x foo` option, `foo` will be added as target of all context stanzas.

### 3.4.1 How Does it Work?

In such a setup, binaries that need to be built and executed in the `default.windows` or `default.android` contexts as part of the build will no longer be executed. Instead, all the binaries that will be executed come from the `default` context. One consequence of this is that all preprocessing (PPX or otherwise) will be done using binaries built in the `default` context.

To clarify this with an example, let's assume that you have the following `src/dune` file:

```
(executable (name foo))  
(rule (with-stdout-to blah (run ./foo.exe)))
```

When building `_build/default/src/blah`, `dune` will resolve `./foo.exe` to `_build/default/src/foo.exe` as expected. However, for `_build/default.windows/src/blah` `dune` will resolve `./foo.exe` to `_build/default/src/foo.exe`

Assuming that the right packages are installed or that your workspace has no external dependencies, Dune will be able to cross-compile a given package without doing anything special.

Some packages might still have to be updated to support cross-compilation. For instance if the `foo.exe` program in the previous example was using `Sys.os_type`, it should instead take it as a command line argument:

```
(rule (with-stdout-to blah (run ./foo.exe -os-type ${os_type})))
```

## 3.5 Dealing with Foreign Libraries

The OCaml programming language can interface with libraries written in foreign languages such as C. This section explains how to do this with Dune. Note that it does not cover how to write the C stubs themselves, but this is covered by the [OCaml manual](#).

More precisely, this section covers:

- How to add C/C++ stubs to an OCaml library
- How to pass specific compilation flags for compiling the stubs
- How to build a library with a foreign build system

In general, Dune has limited support for building source files written in foreign languages. This support is suitable for most OCaml projects containing C stubs, but it is too limited for building complex libraries written in C or other languages. For such cases, Dune can integrate a foreign build system into a normal Dune build.

### 3.5.1 Adding C/C++ Stubs to an OCaml Library

To add C stubs to an OCaml library, simply list the C files without the `.c` extension in the `(foreign_stubs)` field. For instance:

```
(library
 (name mylib)
 (foreign_stubs (language c) (names file1 file2)))
```

You can also add C++ stubs to an OCaml library by specifying `(language cxx)` instead.

Dune is currently not flexible regarding the extension of the C/C++ source files. They have to be `.c` for C files and `.cpp`, `.cc` or `.cxx` for C++ files. If you have source files with other extensions and you want to build them with Dune, you need to rename them first. Alternatively, you can use the *foreign build sandboxing* method described below.

#### Header Files

C/C++ source files may include header files in the same directory as the C/C++ source files or in the same directory group when using `include_subdirs`.

The header files must have the `.h` extension.

For headers that are part of the current project, use `include_dirs` to add an include directory that Dune can track as a dependency:

```
(library
 (name mylib)
 (foreign_stubs
 (language c)
 (names mystubs)
 (include_dirs include)))
```

For headers and libraries installed by the operating system, prefer querying `pkg-config` with `dune-configurator` and passing the discovered compiler and linker flags to Dune. This is useful on systems where libraries are installed in non-standard locations, such as Homebrew prefixes.

```
(library
 (name lz4)
 (foreign_stubs
 (language c)
 (names lz4_stubs)
 (flags (:standard (:include c_flags.sexp))))
 (c_library_flags (:standard (:include c_library_flags.sexp))))
```

The generated `c_flags.sexp` file should contain the C compilation flags, such as `-I` include paths, and `c_library_flags.sexp` should contain link flags, such as `-L` and `-l` flags. See the quick-start section on *defining a library with C stubs using pkg-config* for a complete example.

#### Installing Header Files

It is sometimes desirable to install header files with the library. For that you have two choices: install them explicitly with an `install` stanza or use the `install_c_headers` field of the `library` stanza. This field takes a list of header files names without the `.h` extension. When a library installs header files, they are made visible to users of the library via the include search path.

### 3.5.2 Stub Generation with Dune Ctypes

Beginning in Dune 3.0, it's possible to use the `ctypes` field to generate bindings for C libraries without writing any C code.

Note that Dune support for this feature is experimental and is not subject to backward compatibility guarantees.

To use Dune ctypes stub generation, you must provide two OCaml modules: a “type description” module for describing the C library types and constants, and a “function description” module for describing the C library functions. Additionally, you must list any C headers and a method for resolving build and link flags.

If you're binding a library distributed by your OS, you can use the `pkg-config` utility to resolve any build and link flags. Alternatively, if you're using a locally installed library or a vendored library, you can provide the flags manually.

The “type description” module must define a functor named `Types` with signature `Ctypes.TYPE`. The “function description” module must define a functor named `Functions` with signature `Ctypes.FOREIGN`.

#### A Toy Example

To begin, you must declare the `ctypes` extension in your `dune-project` file:

```
(lang dune 3.25)
(using ctypes 0.3)
```

Next, here is a `dune` file you can use to define an OCaml program that binds a C system library called `libfoo`, which offers `foo.h` in a standard location.

```
(executable
 (name foo)
 (libraries core)
 ; ctypes backward compatibility shims warn sometimes; suppress them
 (flags (:standard -w -9-27))
 (ctypes
  (external_library_name libfoo)
  (build_flags_resolver pkg_config)
  (headers (include "foo.h"))
  (type_description
   (instance Types)
   (functor Type_description))
  (function_description
   (concurrency unlocked)
   (instance Functions)
   (functor Function_description))
  (generated_types Types_generated)
  (generated_entry_point C))
```

This field will introduce a module named `C` into your project, with the sub-modules `Types` and `Functions` that will have your fully-bound C types, constants, and functions.

Given `libfoo` with the C header file `foo.h`:

```
#define FOO_VERSION 1

int foo_init(void);

int foo_fnuar(char *);
```

(continues on next page)

(continued from previous page)

```
void foo_exit(void);
```

Your example `type_description.ml` file is:

```
open Ctypes

module Types (F : Ctypes.TYPE) = struct
  open F

  let foo_version = constant "FOO_VERSION" int
end
```

Your example `function_description.ml` file is:

```
open Ctypes

(* This Types_generated module is an instantiation of the Types
   functor defined in the type_description.ml file. It's generated by
   a C program that Dune creates and runs behind the scenes. *)
module Types = Types_generated

module Functions (F : Ctypes.FOREIGN) = struct
  open F

  let foo_init = foreign "foo_init" (void @-> returning int)

  let foo_fnuvar = foreign "foo_fnuvar" (string_opt @-> returning int)

  let foo_exit = foreign "foo_exit" (void @-> returning void)
end
```

Finally, the entry point of your executable named above, `foo.ml`, demonstrates how to access the bound C library functions and values:

```
let () =
  if (C.Types.foo_version <> 1) then
    failwith "foo only works with libfoo version 1";

  match C.Functions.foo_init () with
  | 0 ->
    C.Functions.foo_fnuvar "fnuvar!";
    C.Functions.foo_exit ()
  | err_code ->
    Printf.eprintf "foo_init failed: %d" err_code;
;;
```

From here, one only needs to run `dune build ./foo.exe` to generate the stubs and build and link the example `foo.exe` program.

Complete information about the `ctypes` combinators used above is available at the [ctypes](#) project.

## Ctypes Field Reference

The `ctypes` field can be used in any `executable(s)` or `library` stanza.

```
((executable|library)
...
(ctypes
  (external_library_name <package-name>)
  (type_description
    (instance <module-name>)
    (functor <module-name>))
  (function_description
    (instance <module-name>)
    (functor <module-name>)
    <optional-function-description-fields>)
  (generated_entry_point <module-name>)
  <optional-ctypes-fields>)
)
```

- `type_description`: the `functor` module is a description of the C library types and constants written in the `ctypes` domain-specific language you wish to bind. The `instance` module is the name of the instantiated functor, inserted into the top-level of the `generated_entry_point` module.
- `function_description`: the `functor` module is a description of the C library functions written in the `ctypes` domain-specific language you wish to bind. The `instance` module is the name of the instantiated functor, inserted into the top-level of the `generated_entry_point` module. The `function_description` field can be repeated. This is useful if you need to specify sets of functions with different concurrency policies (see below).

The instantiated types described above can be accessed from the function descriptions by referencing them as the module specified in optional `generated_types` field.

`<optional-ctypes-fields>` are:

- `(build_flags_resolver <pkg_config|vendored-field>)` tells Dune how to compile and link your foreign library. Specifying `pkg_config` will use the `pkg-config` tool to query the compilation and link flags for `external_library_name`. For vendored libraries, provide the build and link flags using `vendored` field. If `build_flags_resolver` is not specified, the default of `pkg_config` will be used.
- `(generated_types <module-name>)` is the name of an intermediate module. By default, it's named `Types_generated`. You can use this module to access the types defined in `Type_description` from your `Function_description` module(s).
- `(generated_entry_point <module-name>)` is the name of a generated module that your instantiated `Types` and `Functions` modules will instantiated under. We suggest calling it `C`.
- Headers can be added to the generated C files:
  - `(headers (include "include1" "include2" ...))` adds `#include <include1>`, `#include <include2>`. It uses the *Ordered Set Language*.
  - `(headers (preamble <preamble>))` adds directly the preamble. Variables can be used in `<preamble>` such as `#{read: }`.
- Since the Dune's `ctypes` feature is still experimental, it could be useful to add additional dependencies in order to make sure that local headers or libraries are available: `(deps <deps-conf list>)`. See *Dependency Specification* for more details.

`<optional-function-description-fields>` are:

- (`concurrency <sequential|unlocked|lwt_jobs|lwt_preemptive>`) tells `ctypes` stubgen whether to call your C functions with the runtime lock held or released. These correspond to the `concurrency_policy` type in the `ctypes` library. If `concurrency` is not specified, the default of `sequential` will be used.
- (`errno_policy <ignore_errno|return_errno>`) specifies the `errno_policy` passed to the code generator. With `ignore_errno`, the `errno` variable is not accessed or returned by function calls. With `return_errno`, all functions will return the tuple `(retval, errno)`.

`<vendored-field>` is:

- (`vendored (c_flags <flags>) (c_library_flags <flags>)`) provide the build and link flags for binding your vendored code. You must also provide instructions in your dune file on how to build the vendored foreign library; see the `foreign_library` stanza. Usually the `<flags>` should contain `:standard` in order to add the default flags used by the OCaml compiler for C files `use_standard_c_and_cxx_flags`.

### 3.5.3 Foreign Build Sandboxing

When the build of a C library is too complicated to express in the Dune language, it's possible to simply *sandbox* a foreign build. Note that this method can be used to build other things, not just C libraries.

To do that, follow the following procedure:

- Put all the foreign code in a sub-directory
- Tell Dune not to interpret configuration files in this directory via an `data_only_dirs` stanza
- Write a custom rule that:
  - depends on this directory recursively via `source_tree`
  - invokes the external build system
  - copies the generated files
  - the C archive `.a` must be built with `-fpic`
  - the `libfoo.so` must be copied as `dllfoo.so`, and no `libfoo.so` should appear, otherwise the dynamic linking of the C library will be attempted. However, this usually fails because the `libfoo.so` isn't available at the time of the execution.
- *Attach* the C archive files to an OCaml library via *Foreign Archives*.

For instance, let's assume that you want to build a C library `libfoo` using `libfoo`'s own build system and attach it to an OCaml library called `foo`.

The first step is to put the sources of `libfoo` in your project, for instance in `src/libfoo`. Then tell Dune to consider `src/libfoo` as raw data by writing the following in `src/dune`:

```
(data_only_dirs libfoo)
```

The next step is to setup the rule to build `libfoo`. For this, writing the following code `src/dune`:

```
(rule
  (deps (source_tree libfoo))
  (targets libfoo.a dllfoo.so)
  (action
  (no-infer
  (progn
    (chdir libfoo (run make))
    (copy libfoo/libfoo.a libfoo.a)
    (copy libfoo/libfoo.so dllfoo.so))))))
```

We copy the resulting archive files to the top directory where they can be declared as `targets`. The build is done in a *no-infer* action because `libfoo/libfoo.a` and `libfoo/libfoo.so` are dependencies produced by an external build system.

The last step is to attach these archives to an OCaml library as follows:

```
(library
 (name bar)
 (foreign_archives foo))
```

Then, whenever you use the `bar` library, you'll also be able to use C functions from `libfoo`.

### Limitations

When using the sandboxing method, the following limitations apply:

- The build of the foreign code will be sequential
- The build of the foreign code won't be incremental

Both these points could be improved. If you're interested in helping make this happen, please let the Dune team know and someone will guide you.

### Real Example

The `re2` project uses this method to build the `re2` C library. You can look at the file `re2/src/re2_c/dune` in this project to see a full working example.

## 3.5.4 Editor Support for C/C++ Stubs

Added in version 3.23.

Language servers such as `clangd` and `ccls` need a `compile_commands.json` file to provide code navigation, completion, and diagnostics for C and C++ sources. Dune can generate this file for you.

When using `(lang dune 3.23)` or later, `dune build @check` produces a `compilation database` covering all C and C++ sources declared via `foreign_stubs`, `foreign_library`, and `ctypes` stanzas in the workspace. The file is promoted to the workspace root so that editors can find it without configuration. Running `dune clean` removes it. You can also build the file directly with `dune build compile_commands.json`.

Each entry records the full compiler invocation that Dune would use for that source file, including the C compiler, base flags from `ocamlc -config`, user-specified `(flags ...)`, `-I` paths for the OCaml standard library, library dependencies (via `install_c_headers`), and `(include_dirs ...)`.

Since the file is a build artifact promoted to the source tree, you should add `compile_commands.json` to your `.gitignore`.

## 3.6 Generating Documentation

### 3.6.1 Prerequisites

Documentation in Dune is done courtesy of the `odoc` tool. Therefore, to generate documentation in Dune, you will need to install this tool. This should be done with `opam`:

```
$ opam install odoc
```

### 3.6.2 Writing Documentation

Documentation comments will be automatically extracted from your OCaml source files following the syntax described in the section `Text formatting` of the [OCaml manual](#).

Additional documentation pages may be attached to a package using the `documentation` stanza. This stanza attaches `.mld` pages to packages rather than to executables.

### 3.6.3 Building Documentation

Dune's generated documentation is package-oriented. The `@doc` alias builds HTML documentation for public libraries, that is, libraries with a `(public_name ...)`, and for `.mld` pages attached to packages. Packages are declared with `package` stanzas or inferred from `.opam` files.

Executable stanzas are not documented as API pages by `@doc`. A project that contains only executables may therefore build `@doc` successfully without producing package API pages. To produce API documentation, put the documented code in a library with a `(public_name ...)` or attach `.mld` pages to a package.

To generate documentation using `@doc`, build this alias:

```
$ dune build @doc
```

An index page containing links to all the opam packages in your project can be found in:

```
$ open _build/default/_doc/_html/index.html
```

Documentation for private libraries may also be built with `@doc-private`:

```
$ dune build @doc-private
```

But these libraries will not be in the main HTML listing above, since they don't belong to any particular package, but the generated HTML will still be found in `_build/default/_doc/_html/<library>`.

Documentation for public libraries can also be generated as JSON files with the `@doc-json` alias:

```
$ dune build @doc-json
```

These files are produced by `odoc` and can be used by external tools or custom documentation websites.

#### Documentation Stanza: Examples

The `documentation` stanza will attach all the `.mld` files in the current directory in a project with a single package.

```
(documentation)
```

This stanza will attach three `.mld` files to package `foo`. The `.mld` files should be named `foo.mld`, `bar.mld`, and `baz.mld`

```
(documentation
(package foo)
(mld_files foo bar baz))
```

This stanza will attach all `.mld` files to the inferred package, excluding `wip.mld`, in the current directory:

```
(documentation
(mld_files :standard \ wip))
```

All `.mld` files attached to a package will be included in the generated `.install` file for that package. They'll be installed by `opam`.

```
(documentation
 (files
  (glob_files_rec
   (doc/* with_prefix .))))
```

All files in the `doc/` folder will be attached to the inferred package. The hierarchy between them will be preserved, relative to `doc/` considered as the root.

### Note

`dune` does not yet support building the documentation with a non-flat hierarchy, or with non-`mld` files. However, it supports installing those files following a convention, so that `odoc_driver` can build the docs with hierarchy and asset files.

## Package Entry Page

The `index.mld` file (specified as `index` in `mld_files`) is treated specially by Dune. This will be the file used to generate the entry page for the package, linked from the main package listing.

To generate pleasant documentation, we recommend writing an `index.mld` file with at least short description of your package and possibly some examples.

If you do not write your own `index.mld` file, Dune will generate one with the entry modules for your package. But this generated file will not be installed.

## 3.6.4 Passing Options to `odoc`

```
(env
 (<profile>
  (odoc <optional-fields>)))
```

See `env` for more details on the `(env ...)` stanza. `<optional-fields>` are:

- `(warnings <mode>)` specifies how warnings should be handled. `<mode>` can be: `fatal` or `nonfatal`. The default value is `nonfatal`. This field is available since Dune 2.4.0 and requires `odoc 1.5.0`.

## 3.6.5 Local Documentation Search Using Sherlodoc

If Sherlodoc is installed, generated HTML documentation will include a search bar. It supports search by name, documentation and fuzzy type search.

It can be installed with:

```
$ opam install sherlodoc
```

## 3.7 How to Load Additional Files at Runtime

There are many ways for applications to load files at runtime and Dune provides a well-tested, key-in-hand portable system for doing so. The Dune model works by defining `sites` where files will be installed and looked up at runtime. At runtime, each site is associated to a list of directories which contain the files added in the site.

**WARNING:** This feature remains experimental and is subject to breaking changes without warning. It must be explicitly enabled in the `dune-project` file with `(using dune_site 0.1)`

### 3.7.1 Sites

#### Defining a Site

A site is defined in a package *package* in the `dune-project` file. It consists of a name and a *section* (e.g `lib`, `share`, etc) where the site will be installed as a sub-directory.

```
(lang dune 3.25)
(using dune_site 0.1)
(name mygui)

(package
 (name mygui)
 (sites (share themes)))
```

#### Adding Files to a Site

Here the package `mygui` defines a site named `themes` that will be located in the section `share`. This package can add files to this site using the *install stanza*:

```
(install
 (section (site (mygui themes)))
 (files
 (layout.css as default/layout.css)
 (ok.png as default/ok.png)
 (ko.png as default/ko.png)))
```

Another package `mygui_material_theme` can install files inside `mygui` directory for adding a new theme. Inside the scope of `mygui_material_theme` the `dune` file contains:

```
(install
 (section (site mygui themes))
 (files
 (layout.css as material/layout.css)
 (ok.png as material/ok.png)
 (ko.png as material/ko.png)))
```

The package `mygui` must be present in the workspace or installed.

#### Warning

Two files should not be installed by different packages at the same destination.

#### Getting the Locations of a Site at Runtime

The executable `mygui` will be able to get the locations of the `themes` site using the *generate\_sites\_module stanza*.

```
(executable
 (name mygui)
 (modules mygui mysites)
 (libraries dune-site))

(generate_sites_module
```

(continues on next page)

```
(module mysites)
(sites mygui))
```

The generated module `mysites` depends on the library `dune-site` provided by Dune. As such, the dependency on `dune-site` must be specified explicitly.

### Warning

An executable that depends, even transitively, on `dune-site` or `dune-site.plugins` will be compiled with `-linkall`, regardless of other options.

### Note

The dependency on `dune-site` also needs to be added to the `depends` field of your project. If left out it might be accidentally captured via transitive dependencies, however this will not work if `implicit_transitive_deps` is set to `false` in the `dune-project` file. It is always recommended to specify `dune-site` as a dependency explicitly.

Then inside `mygui.ml` module the locations can be recovered and used:

```
(** Locations of the site for the themes *)
let themes_locations : string list = Mysites.Sites.themes

(** Merge the contents of the directories in [dirs] *)
let lookup_dirs dirs =
  List.filter Sys.file_exists dirs
  |> List.map (fun dir -> Array.to_list (Sys.readdir dir))
  |> List.concat

(** Get the available themes *)
let find_available_themes () = lookup_dirs themes_locations

(** [lookup_file name dirs] finds the first file called [name] in [dirs] *)
let lookup_file filename dirs =
  List.find_map
    (fun dir ->
      let filename' = Filename.concat dir filename in
      if Sys.file_exists filename' then Some filename' else None)
    dirs

(** [lookup_theme_file theme file] get the [file] of the [theme] *)
let lookup_theme_file file theme =
  lookup_file (Filename.concat theme file) themes_locations

let get_layout_css = lookup_theme_file "layout.css"
let get_ok_ico = lookup_theme_file "ok.png"
let get_ko_ico = lookup_theme_file "ko.png"
```

## Tests

During tests, the files are copied into the sites through the dependency (package `mygui`) and (package `mygui_material_theme`) as for other files in install stanza.

## Installation

Installation is done simply with `dune install`; however, if one wants to install this tool to make it relocatable, one can use `dune install --relocatable --prefix $dir`. The files will be copied to the directory `$dir` but the binary `$dir/bin/mygui` will find the site location relative to its location. So even if the directory `$dir` is moved, `themes_locations` will be correct.

For installation through `opam`, `dune install` must be invoked with the option `--create-install-files` which creates an install file `<pkg>.install` and copy the file that needs substitution to an intermediary directory. The `<pkg>.opam` file generated by Dune *generate\_opam\_files* does the right invocation.

## Implementation Details

The main difficulty for sites is that their directories are found at different locations at different times:

- When the package is available locally, the location is inside `_build`
- When the package is installed, the location is inside the install prefix
- If a local package wants to install files to the site of another installed package the location is at the same time in `_build` and in the install prefix of the second package.

With the last example, we see that the location of a site is not always a single directory, but rather it can consist of a sequence of directories: `["dir1" ; "dir2"]`. So a lookup must first look into `dir1`, then into `dir2`.

### 3.7.2 Plugins and Dynamic Loading of Packages

Dune allows you to define and load plugins without having to deal with specific compilation, installation directories, dependencies, or the `Dynlink_` module.

To define a plugin:

- The package defining the plugin interface must define a *site* where the plugins must live. Traditionally, this is in `(lib plugins)`, but it's just a convention.
- Define a library that each plugin must use to register itself (or otherwise provide its functionality).
- Define the plugin in another package using the *plugin* stanza.
- Generate a module that may load all available plugins using the *generated\_module* stanza.

## Example

We demonstrate an example of the scheme above. The example consists of the following components:

Inside package *app*:

- An executable *app*, that we intend to extend with plugins
- A library *app.registration* which defines the plugin registration interface
- A generated module *Sites* which can load available plugins at runtime
- An executable *app* that will use the module *Sites* to load all the plugins

Inside package *Plugin1*, we declare a plugin using the *app.registration* api and the *plugin* stanza.

### Directory structure

```
.
├── app.ml
├── dune
├── dune-project
├── plugin
│   ├── dune
│   ├── dune-project
│   └── plugin1_impl.ml
└── registration.ml
```

### Main Executable (C)

- The dune-project file:

```
(lang dune 3.25)
(using dune_site 0.1)
(name app)

(package
  (name app)
  (sites (lib plugins)))
```

- The dune file:

```
(executable
  (public_name app)
  (modules sites app)
  (libraries app.register dune-site dune-site.plugins))

(library
  (public_name app.register)
  (name registration)
  (modules registration))

(generate_sites_module
  (module sites)
  (plugins (app plugins)))
```

The generated module *sites* depends here also on the library *dune-site.plugins* because the *plugins* optional field is requested.

If the executable being created is an OCaml toplevel, then the `libraries` stanza needs to also include the `dune-site.toplevel` library. This causes the loading to use the toplevel's normal loading mechanism rather than `Dynload.loadfile` (which is not allowed in toplevels).

- The module `registration.ml` of the library `app.registration`:

```
let todo : (unit -> unit) Queue.t = Queue.create ()
```

- The code of the executable `app.ml`:

```
(* load all the available plugins *)
let () = Sites.Plugins.Plugins.load_all ()
```

(continues on next page)

(continued from previous page)

```
let () = print_endline "Main app starts..."
(* Execute the code registered by the plugins *)
let () = Queue.iter (fun f -> f ()) Registration.todo
```

### The Plugin “plugin1”

- The plugin/dune-project file:

```
(lang dune 3.25)
(using dune_site 0.1)

(generate_opam_files true)

(package
  (name plugin1))
```

- The plugin/dune file:

```
(library
  (public_name plugin1.plugin1_impl)
  (name plugin1_impl)
  (modules plugin1_impl)
  (libraries app.register))

(plugin
  (name plugin1)
  (libraries plugin1.plugin1_impl)
  (site (app plugins)))
```

- The code of the plugin plugin/plugin1\_impl.ml:

```
let () =
  print_endline "Registration of Plugin1";
  Queue.add (fun () -> print_endline "Plugin1 is doing something...") Registration.todo
```

### Running the Example

```
$ dune build @install && dune exec ./app.exe
Registration of Plugin1
Main app starts...
Plugin1 is doing something...
```

## 3.8 Instrumentation

In this section, we’ll explain how to define and use instrumentation backends (such as `bisect_ppx` or `landmarks`) so that you can enable and disable coverage via `dune-workspace` files or by passing a command-line flag or environment variable. In addition to providing an easy way to toggle instrumentation of your code, this setup avoids creating a hard dependency on the precise instrumentation backend in your project.

### 3.8.1 Specifying What to Instrument

When an instrumentation backend is activated, Dune will only instrument libraries and executables for which the user has requested instrumentation.

To request instrumentation, one must add the following field to a library or executable stanza:

```
(library
 (name ...)
 (instrumentation
  (backend <name> <args>)
  <optional-fields>))
```

The backend <name> can be passed into arguments using <args>.

This field can be repeated multiple times in order to support various backends. For instance:

```
(library
 (name foo)
 (modules foo)
 (instrumentation (backend bisect_ppx --bisect-silent yes))
 (instrumentation (backend landmarks)))
```

This will instruct Dune that when either the `bisect_ppx` or `landmarks` instrumentation is activated, the library should be instrumented with this backend.

By default, these fields are simply ignored; however, when the corresponding instrumentation backend is activated, Dune will implicitly add the relevant ppx rewriter to the list of ppx rewriters.

At the moment, it isn't possible to instrument code that's preprocessed via an action preprocessors. As these preprocessors are quite rare nowadays, there is no plan to add support for them in the future.

<optional-fields> are:

- `(deps <deps-conf list>)` specifies extra instrumentation dependencies, for instance, if it reads a generated file. The dependencies are only applied when the instrumentation is actually enabled. The specification of dependencies is described in *Dependency Specification*.

### 3.8.2 Enabling/Disabling Instrumentation

Activating an instrumentation backend can be done via the command line or the `dune-workspace` file.

Via the command line, it is done as follows:

```
$ dune build --instrument-with <names>
```

Here <names> is a comma-separated list of instrumentation backends. For example:

```
$ dune build --instrument-with bisect_ppx,landmarks
```

This will instruct Dune to activate the given backend globally, i.e., in all defined build contexts.

It's also possible to enable instrumentation backends via the `dune-workspace` file, either globally or for specific build contexts.

To enable an instrumentation backend globally, type the following in your `dune-workspace` file:

```
(lang dune 3.25)
(instrument_with bisect_ppx)
```

or for each context individually:

```
(lang dune 3.25)
(context default)
(context (default (name coverage) (instrument_with bisect_ppx)))
(context (default (name profiling) (instrument_with landmarks)))
```

If both the global and local fields are present, the precedence is the same as the `profile` field: the per-context setting takes precedence over the command-line flag, which takes precedence over the global field.

### 3.8.3 Declaring an Instrumentation Backend

Instrumentation backends are libraries with the special field (`instrumentation.backend`). This field instructs Dune that the library can be used as an instrumentation backend, and it also provides the parameters specific to this backend.

Currently, Dune will only support `ppx` instrumentation tools, and the instrumentation library must specify the `ppx` rewriters that instruments the code. This can be done as follows:

```
(library
  ...
  (instrumentation.backend
    (ppx <ppx-rewriter-name>)))
```

When such an instrumentation backend is activated, Dune will implicitly add the mentioned `ppx` rewriter to the list of `ppx` rewriters for libraries and executables that specify this instrumentation backend.

## 3.9 JavaScript Compilation With `Js_of_ocaml`

`Js_of_ocaml` is a compiler from OCaml to JavaScript. The compiler works by translating OCaml bytecode to JS files. The compiler can be installed with `opam`:

```
$ opam install js_of_ocaml-compiler
```

### 3.9.1 Compiling to JS

Dune has full support building `js_of_ocaml` libraries and executables transparently. There's no need to customize or enable anything to compile OCaml libraries/executables to JS.

To build a JS executable, just define an executable as you would normally. Consider this example:

```
$ echo 'print_endline "hello from js"' > foo.ml
```

With the following dune file:

```
(executable (name foo) (modes js))
```

And then request the `.js` target:

```
$ dune build ./foo.bc.js
$ node _build/default/foo.bc.js
hello from js
```

Similar targets are created for libraries, but we recommend sticking to the executable targets.

If you're using the `js_of_ocaml` syntax extension, you must remember to add the appropriate PPX in the `preprocess` field:

```
(executable
  (name foo)
  (modes js)
  (preprocess (pps js_of_ocaml-ppx)))
```

### 3.9.2 Separate Compilation

Dune supports two modes of compilation:

- Direct compilation of a bytecode program to JavaScript. This mode allows `js_of_ocaml` to perform whole-program deadcode elimination and whole-program inlining.
- Separate compilation, where compilation units are compiled to JavaScript separately and then linked together. This mode is useful during development as it builds more quickly.

The separate compilation mode will be selected when the build profile is `dev`, which is the default. It can also be explicitly specified in an `env` stanza (see *env*) or per executable inside (`js_of_ocaml (compilation_mode ...)`) (see *executable*)

### 3.9.3 Sourcemap

`Js_of_ocaml` can generate sourcemap for the generated JavaScript file. It can either embed it at the end of the `.js` file or write it to separate file. By default, it is inlined when using the `dev` build profile and is not generated otherwise. The behavior can explicitly be specified in an `env` stanza (see *env*) or per executable inside (`js_of_ocaml (sourcemap ...)`) (see *executable*)

## 3.10 Wasm Compilation With `Wasm_of_ocaml`

`Wasm_of_ocaml` is a compiler from OCaml to WebAssembly (Wasm for short). The compiler works by translating OCaml bytecode to Wasm code. The compiler can be installed with `opam`:

```
$ opam install wasm_of_ocaml-compiler
```

### 3.10.1 Compiling to Wasm

Dune has full support for building `wasm_of_ocaml` libraries and executables transparently. There's no need to customise or enable anything to compile OCaml libraries/executables to Wasm.

To build a Wasm executable, just define an executable as you would normally. Consider this example:

```
$ echo 'print_endline "hello from wasm"' > foo.ml
```

With the following dune file:

```
(executable (name foo) (modes wasm))
```

And then request the `.wasm.js` target:

```
$ dune build ./foo.bc.wasm.js
$ node _build/default/foo.bc.wasm.js
hello from wasm
```

If you're using the `js_of_ocaml` syntax extension, you must remember to add the appropriate PPX in the `preprocess` field:

```
(executable
  (name foo)
  (modes wasm)
  (preprocess (pps js_of_ocaml-ppx)))
```

### 3.10.2 Selective Compilation

The `js` and `wasm` modes can be selectively disabled using the `(js_of_ocaml (enabled_if ...))` and `(wasm_of_ocaml (enabled_if ...))` options. This allows for instance to generate one, or the other dependings on a profile:

```
(env
  (js-only
    (wasm_of_ocaml
      (enabled_if false)))
  (wasm-only
    (js_of_ocaml
      (enabled_if false))))
```

To be able to invoke the generated code using the same JavaScript script name in all cases, you can add a rule to copy the Wasm launcher script when the `js_of_ocaml` compilation is disabled.

```
(rule
  (action
    (copy foo.bc.wasm.js foo.bc.js))
  (enabled_if
    (= %{{profile}} wasm-only)))
```

### 3.10.3 Separate Compilation

Dune supports two modes of compilation:

- Direct compilation of a bytecode program to Wasm. This mode allows `wasm_of_ocaml` to perform whole-program deadcode elimination and whole-program inlining.
- Separate compilation, where compilation units are compiled to Wasm separately and then linked together. This mode is useful during development as it builds more quickly.

The separate compilation mode will be selected when the build profile is `dev`, which is the default. It can also be explicitly specified in an `env` stanza (see *env*) or per executable inside `(wasm_of_ocaml (compilation_mode ...))` (see *executable*)

### 3.10.4 Sourcemap

`Wasm_of_ocaml` can generate sourcemaps for the generated Wasm code. By default, they are generated when using the `dev` build profile and are not generated otherwise. The behavior can explicitly be specified in an `env` stanza (see *env*) or per executable inside `(wasm_of_ocaml (sourcemap ...))` (see *executable*)

## 3.11 JavaScript Compilation With Melange

### 3.11.1 Introduction

`Melange` compiles OCaml to JavaScript. It produces one JavaScript file per OCaml module. `Melange` can be installed with `opam`:

```
$ opam install melange
```

Dune can build Melange projects, and produces JavaScript files by defining a `melange.emit` stanza. Dune libraries may also define Melange libraries by adding `melange` to `(modes ...)` in the `library` stanza.

Melange support must be enabled in the `dune-project` file:

```
(using melange 1.0)
```

Once that's in place, you can use the Melange mode in `library` and `melange.emit` stanzas.

### 3.11.2 Simple Project

Let's start by looking at a simple project with Melange and Dune. Subsequent sections explain the different concepts used here in further detail.

First, make sure that the `dune-project` file specifies at least version 3.20 of the Dune language, and the Melange extension is enabled:

```
(lang dune 3.25)
(using melange 1.0)
```

Next, write a `dune` file with a `melange.emit` stanza:

```
(melange.emit
 (target output))
```

Finally, add a source file to build:

```
$ echo 'Js.log "hello from melange"' > hello.ml
```

After running `dune build @melange` or just `dune build`, Dune produces the following file structure:

```
.
├── _build
│   ├── default
│   │   └── output
│   │       └── hello.js
├── dune
├── dune-project
└── hello.ml
```

The resulting JavaScript can now be run:

```
$ node _build/default/output/hello.js
hello from melange
```

### 3.11.3 Libraries

Adding Melange support to Dune libraries is done as follows:

- `(modes melange)`: adding `melange` to `modes` is required. This field also supports the *Ordered Set Language*.
- `(melange.runtime_deps <deps>)`: optionally, define any runtime dependencies using `melange.runtime_deps`. This field is analog to the `runtime_deps` field used in `melange.emit` stanzas.

- (`melange.compile_flags <flags>`): optionally, pass flags to `melc` compiler. This field is analogous to the `compile_flags` field used in `melange.emit` stanzas.

### 3.11.4 `melange.emit`

Added in version 3.8.

The `melange.emit` stanza produces JavaScript files from Melange libraries or entry-point modules. It's similar to the OCaml *executable* stanza, with the exception that there is no linking step.

```
(melange.emit
 (target <target>)
 <optional-fields>)
```

- `<target>` is the name of the folder inside the build directory where Dune will compile the resulting JavaScript. In particular, the folder will be placed under `_build/default/$path-to-directory-of-melange-emit-stanza`.

**Note:** when using *promotion*, Dune will additionally copy the resulting JavaScript back to the source tree, next to the original source files.

`$path-to-directory-of-melange-emit-stanza` matches the file structure of the source tree. For example, given the following source tree:

```
├─ dune # (melange.emit (target output) (libraries lib))
├─ app.ml
├─ lib
│   └─ dune # (library (name lib) (modes melange))
│       └─ helper.ml
```

The resulting layout in `_build/default/output` will be as follows:

```
output
├─ app.js
├─ lib
│   └─ lib.js
│       └─ helper.js
```

`<optional-fields>` are:

- (`alias <alias-name>`) specifies an alias to which to attach the targets of the `melange.emit` stanza.
  - These targets include the `.js` files generated by the stanza modules, the targets for the `.js` files of any library that the stanza depends on, and any copy rules for runtime dependencies (see `runtime_deps` field below).
  - By default, all stanzas will have their targets attached to an alias `melange`. The behavior of this default alias is exclusive: if an alias is explicitly defined in the stanza, the targets from this stanza will be excluded from the `melange` alias.
  - The targets of `melange.emit` are also attached to the Dune default alias (`@all`), regardless of whether the (`alias ...`) field is present.
- (`module_systems <module_systems>`) specifies the JavaScript import and export format used. The values allowed for `<module_systems>` are `es6` and `commonjs`.
  - `es6` will follow [JavaScript modules](#), and will produce `import` and `export` statements.
  - `commonjs` will follow [CommonJS modules](#), and will produce `require` calls and export values with `module.exports`.

- If no extension is specified, the resulting JavaScript files will use `.js`. You can specify a different extension with a pair (`<module_system> <extension>`), e.g. `(module_systems (es6 mjs))`.
- Multiple module systems can be used in the same field as long as their extensions are different. For example, `(module_systems commonjs (es6 mjs))` will produce one set of JavaScript files using CommonJS and the `.js` extension, and another using ES6 and the `.mjs` extension.
- `(modules <modules>)` specifies what modules will be built with Melange. By default, if this field is not defined, Dune will use all the `.ml/.re` files in the same directory as the `dune` file. This includes module sources present in the file system as well as modules generated by user rules. You can restrict this list by using an explicit `(modules <modules>)` field. `<modules>` uses the *Ordered Set Language*, where elements are module names and don't need to start with an uppercase letter. For instance, to exclude module `Foo`, use `(modules :standard \ foo)`.
- `(libraries <library-dependencies>)` specifies Melange library dependencies. Melange libraries can only use the simple form, like `(libraries foo pkg.bar)`. Keep in mind the following limitations:
  - The `re_export` form is not supported.
  - All the libraries included in `<library-dependencies>` have to support the `melange` mode (see the section about libraries below).
- `(package <package>)` allows the user to define the JavaScript package to which the artifacts produced by the `melange.emit` stanza will belong.
- `(runtime_deps <paths-to-deps>)` specifies dependencies that should be copied to the build folder together with the `.js` files generated from the sources. These runtime dependencies can include assets like CSS files, images, fonts, external JavaScript files, etc. `runtime_deps` adhere to the formats in *Dependency Specification*. For example `(runtime_deps ./path/to/file.css (glob_files_rec ./fonts/*))`.
- `(emit_stdlib <bool>)` allows the user to specify whether the Melange standard library should be included as a dependency of the stanza or not. The default is `true`. If this option is `false`, the Melange standard library and runtime JavaScript files won't be produced in the target directory.
- `(promote <options>)` promotes the generated `.js` files to the source tree. The options are the same as for the *rule promote mode*. Adding `(promote (until-clean))` to a `melange.emit` stanza will cause Dune to copy the `.js` files to the source tree and `dune clean` to delete them. Check *Promotion* for more details.
- `(preprocess <preprocess-spec>)` specifies how to preprocess files when needed. The default is `no_preprocessing`. Additional options are described in the *Preprocessing Specification* section.
- `(lint <preprocess-spec>)` specifies how to lint source files when building the `@lint` alias. The default is `no_preprocessing`. The syntax is described in *Linting*.
- `(preprocessor_deps (<deps-conf list>))` specifies extra preprocessor dependencies, e.g., if the preprocessor reads a generated file. The dependency specification is described in the *Dependency Specification* section.
- `(compile_flags <flags>)` specifies compilation flags specific to `melc`, the main Melange executable. `<flags>` is described in detail in the *Ordered Set Language* section. It also supports `(:include ...)` forms. The value for this field can also be taken from `env` stanzas. It's therefore recommended to add flags with e.g. `(compile_flags :standard <my options>)` rather than replace them.
- `(root_module <module>)` specifies a `root_module` that collects all listed dependencies in `libraries`. See the documentation for `root_module` in the *library* stanza.
- `(allow_overlapping_dependencies)` is the same as the corresponding field of *library*.
- `(enabled_if <blang expression>)` conditionally disables a `melange emit` stanza. The JavaScript files associated with the stanza won't be built. The condition is specified using the *Boolean Language*.

### 3.11.5 Recommended Practices

#### Keep Bundles Small by Reducing the Number of `melange.emit` Stanzas

It is recommended to minimize the number of `melange.emit` stanzas that a project defines: using multiple `melange.emit` stanzas will cause multiple copies of the JavaScript files to be generated if the same libraries are used across them. As an example:

```
(melange.emit
 (target app1)
 (libraries foo))

(melange.emit
 (target app2)
 (libraries foo))
```

The JavaScript artifacts for library `foo` will be emitted twice in the `_build` folder. They will be present under `_build/default/app1` and `_build/default/app2`.

This can have unexpected impact on bundle size when using tools like Webpack or Esbuild, as these tools will not be able to see shared library code as such, as it would be replicated across the paths of the different stanzas `target` folders.

#### Faster Builds With `subdir` and `dirs` Stanzas

Melange libraries might be installed from the `npm` package repository, together with other JavaScript packages. To avoid having Dune inspect unnecessary folders in `node_modules`, it is recommended to explicitly include only the folders that are relevant for Melange builds.

This can be accomplished by combining `subdir` and `dirs` stanzas in a `dune` file next to the `node_modules` folder. The `vendored_dirs` stanza can be used to avoid warnings in Melange libraries during the application build. The `data_only_dirs` stanza can be useful as well if you need to override the build rules in one of the packages.

```
(subdir
 node_modules
 (vendored_dirs reason-react)
 (dirs reason-react))
```

### 3.11.6 Promotion

Compiling and promoting Melange output in Dune is slightly different than compiling OCaml:

- Limitations in Dune `rule production` require a *target directory* in `melange.emit`.
  - The target directory is *total*: it can be exported as is from the Dune build directory
- Many popular tools and frameworks in the JavaScript ecosystem today rely on convention over configuration, especially as it relates to folder structure. When using *promotion*

### 3.11.7 Design choices

Melange support in Dune follows the following design choices:

- `melange.emit` produces a “total” directory: the artifacts in the `target` directory contain all the JavaScript and `runtime_deps` assets necessary to run the application either through a JS framework, a bundler, or otherwise a deployment (excluding external dependencies installed via a JS package manager). The structure is designed such that relative paths and dependencies work out of the box relative to their paths in the source tree, before compilation.

- public libraries are compiled to `%{target}/node_modules/%{lib_name}` such that the [resolution algorithm](#) works to resolve Melange libraries from compiled JS code.
- JavaScript output is promoted to the source tree

### 3.12 Virtual Libraries

Virtual libraries correspond to Dune’s ability to compile parameterised libraries and delay the selection of concrete implementations until linking an executable.

The feature introduces two kinds of libraries: virtual and implementations. A *virtual library* corresponds to an interface (although it may contain partial implementation). An *implementation* of a virtual library fills in all unimplemented modules in the virtual library.

The benefit of this partition is that other libraries may depend on and compile against the virtual library, and they might only select concrete implementations for these virtual libraries when linking executables. An example where this might be useful would be a virtual, cross-platform, `clock` library. This library would have `clock.unix` and `clock.win` implementations. Executable using `clock` or libraries that use `clock` would conditionally select one of the implementations, depending on the target platform.

#### Note

This feature is sometimes informally called “variants”. However, this term refers to a related feature that has been removed in Dune 2.6 in favor of default implementation, and the correct term for the mechanism as a whole is “virtual libraries”.

#### 3.12.1 Virtual Library

To define a virtual library, a `virtual_modules` field must be added to an ordinary library stanza, and the version of the Dune language must be at least 1.5. This field defines modules for which only an interface would be present (mli only):

```
(library
  (name clock)
  ;; clock.mli must be present, but clock.ml must not be
  (virtual_modules clock))
```

Apart from this field, the virtual library is defined just like a normal library and may use all the other fields. A virtual library may include other modules (with or without implementations), which is why it’s not a pure “interface” library.

#### Note

The `virtual_modules` field is not merged in `modules`, which represents the total set of modules in a library. If a directory has more than one stanza and thus a `modules` field must be specified, virtual modules still need to be added in `modules`.

#### 3.12.2 Implementation

An implementation for a library is defined as:

```
(library
  (name clock_unix)
```

(continues on next page)

(continued from previous page)

```
;; clock.ml must be present, but clock.mli must not be
(implements clock))
```

The name field is slightly different for an implementation than it is for a normal library. The name is just an internal name to refer to the implementation, it doesn't correspond to any particular module like it does in the virtual library.

Other libraries may then depend on the virtual library as if it was a regular library:

```
(library
 (name calendar)
 (libraries clock))
```

But when it comes to creating an executable, we must now select a valid implementation for every virtual library that we've used:

```
(executable
 (name birthday-reminder)
 (libraries
  clock_unix ;; leaving this dependency will make dune loudly complain
  calendar))
```

### 3.12.3 Default Implementation

A virtual library may select a default implementation, which is enabled after variant resolution if no suitable implementation has been found.

```
(library
 (name time)
 (virtual_modules time)
 (default_implementation time-js))
```

The default implementation must live in the same package as the virtual library. In the example above, that would mean that the `time-js` and `time` libraries must be in the same package

### 3.12.4 Limitations

The current implementation of virtual libraries suffers from a few limitations. Some of these are temporary.

- It's impossible to link more than one implementation for the same virtual library in one executable.
- It's not possible for implementations to introduce new public modules. That is, modules that aren't a part of the virtual library's CMI. Consequently, a module in an implementation either implements a virtual module or is private.
- It isn't possible to load virtual libraries into `utop`. As a result, any directory that contains a virtual library will not work with `$ dune utop`. This is an essential limitation, but it would be best to somehow skip these libraries or provide an implementation for them when loading a toplevel.
- Virtual libraries must be defined using Dune. It's not possible for Dune to implement virtual libraries created outside of Dune. On the other hand, virtual libraries and implementations defined using Dune should be usable with `findlib`-based build systems.
- It's impossible for a library to be both virtual and implement another library. This isn't very useful, but it could technically be used to create partial implementations. It is possible to lift this restriction if there's enough demand.

## 3.13 Writing and Running Tests

Dune tries to streamline the testing story as much as possible, so you can focus on the tests themselves and not bother with setting up various test frameworks.

In this section, we'll explain the workflow to deal with tests in Dune. In particular, we'll see how to run the test suite of a project, how to describe your tests to Dune, and how to promote test results as expectation.

We distinguish three kinds of tests:

- Inline tests - written directly inside the `.ml` files of a library
- Custom tests - run an executable, possibly followed by an action such as diffing the produced output.
- Cram tests - expect tests written in [Cram](#) style.

### 3.13.1 Running Tests

Whatever the tests of a project are, the usual way to run tests with Dune is to call `dune runtest` from the shell. `dune test` is an alias for the same command. Without arguments, this runs the tests defined in the current directory and any subdirectory recursively.

`dune runtest` and `dune test` are not exactly the same as `dune build @runtest`. The `runtest` command treats its arguments as test paths, defaulting to `.` when none are supplied, so it can run all tests in a directory, one individual test, or several selected tests.

Most projects use Dune's implicit default *build context* and do not need to think about contexts when running tests. If a workspace defines non-default *workspace contexts*, `dune runtest` chooses the context named `default`, or the only available context when there is just one. If a workspace defines multiple contexts and none of them is named `default`, pass `--context` or a path under a *build context root*, such as `_build/<context>`.

By contrast, `dune build @runtest` builds the `runtest` alias as a normal build target. In a workspace with multiple contexts, an unqualified alias target is built in every configured context. This form is useful when you want to run tests in conjunction with other targets. For instance:

```
$ dune build @install @runtest
$ dune build @install @test/runtest
```

This is distinct from plain `dune build`, which builds the *default alias*. By default, that alias recursively builds *all* file targets, including test executables and generated files in test directories, but it doesn't run actions attached only to `runtest`. If you want plain `dune build` to skip a tests subtree, override `default` in a parent directory.

#### Running a Single Test

If you would only like to run a single test for your project, you may use `dune exec` to run the test executable (for the sake of this example, `project/tests/myTest.ml`):

```
$ dune exec project/tests/myTest.exe
```

To run *Cram Tests* you can pass their paths to the `dune test` command.

```
$ dune test tests/myCramTest.t
```

This works both for directory and file cram tests.

## Running Tests in a Directory

You can also pass a directory argument to run the tests from a subtree. For instance, `dune runtest test` will only run the tests from the `test` directory and any subdirectory of `test` recursively.

### 3.13.2 Inline Tests

There are several inline tests frameworks available for OCaml, such as `ppx_inline_test` and `qtest`. We will use `ppx_inline_test` as an example because it has the necessary setup to be used with Dune out of the box.

`ppx_inline_test` allows one to write tests directly inside `.ml` files as follows:

```
let rec fact n = if n = 1 then 1 else n * fact (n - 1)

let%test _ = fact 5 = 120
```

The file must be preprocessed with the `ppx_inline_test` PPX rewriter, so for instance the dune file might look like this:

```
(library
 (name foo)
 (preprocess (pps ppx_inline_test)))
```

In order to tell Dune that our library contains inline tests, we have to add an `inline_tests` field:

```
(library
 (name foo)
 (inline_tests)
 (preprocess (pps ppx_inline_test)))
```

We can now build and execute this test by running `dune runtest`. For instance, if we make the test fail by replacing `120` by `0` we get:

```
$ dune runtest
[...]
File "src/fact.ml", line 3, characters 0-25: <<(fact 5) = 0>> is false.

FAILED 1 / 1 tests
```

Every inline test library generates an alias with the library name prefixed by `runtest-`. You can build the specific inline test library by running `dune build @runtest-foo` in this case.

Note that in this case Dune knew how to build and run the tests without any special configuration. This is because `ppx_inline_test` defines an inline tests backend that's used by the library. Some other frameworks, such as `qtest`, don't have any special library or PPX rewriter. To use such a framework, you must tell Dune about it, as it cannot guess. You can do that by adding a backend field:

```
(library
 (name foo)
 (inline_tests (backend qtest.lib)))
```

In the example above, the name `qtest.lib` comes from the `public_name` field in `qtest`'s own `dune` file.

Note that using `ppx_inline_test` requires that the opam package `ppx_inline_test` be installed in your switch. If you use `ppx_inline_test` in a package then that package must *unconditionally* depend on `ppx_inline_test` (ie. `ppx_inline_test` can't be a `with-test` dependency).

### Inline Expectation Tests

Inline expectation tests are a special case of inline tests where written OCaml code prints something followed by what you expect this code to print. For instance, using `ppx_expect`:

```
let%expect_test _ =
  print_endline "Hello, world!";
  [%expect{|
    Hello, world!
  |}]
```

The test procedure consist of executing the OCaml code and replacing the contents of the `[%expect]` extension point by the real output. You then get a new file that you can compare to the original source file. Expectation tests are a neat way to write tests as the following test elements are clearly identified:

- The code of the test
- The test expectation
- The test outcome

You can have a look at [this blog post](#) to find out more about expectation tests. To Dune, the workflow for expectation tests is always as follows:

- Write the test with some empty expect nodes in it
- Run the tests
- Check the suggested correction and promote it as the original source file if you are happy with it

Dune makes this workflow very easy. Simply add `ppx_expect` to your list of PPX rewriters as follows:

```
(library
 (name foo)
 (inline_tests)
 (preprocess (pps ppx_expect)))
```

Then calling `dune runtest` will run these tests, and in case of mismatch, Dune will print a diff of the original source file and the suggested correction. For instance:

```
$ dune runtest
[...]
-src/fact.ml
+src/fact.ml.corrected
File "src/fact.ml", line 5, characters 0-1:
let rec fact n = if n = 1 then 1 else n * fact (n - 1)

let%expect_test _ =
  print_int (fact 5);
- [%expect]
+ [%expect{| 120 |}]
```

In order to accept the correction, simply run:

```
$ dune promote
```

You can also make Dune automatically accept the correction after running the tests by typing:

```
$ dune runtest --auto-promote
```

Finally, some editor integration can make the editor do the promotion, which in turn makes the workflow even smoother.

### Running a Subset of the Test Suite

You may also run a group of tests located under a directory with:

```
$ dune runtest mylib/tests
```

The above command will run all tests defined in tests and its subdirectories.

### Running Tests in Bytecode or JavaScript

By default, Dune runs inline tests in native mode, unless native compilation isn't available. In which case, it runs them in bytecode. You can change this setting to choose the modes that tests should run in. To do this, add a `modes` field to the `inline_tests` field. Available modes are:

- `byte` for running tests in byte code
- `native` for running tests in native mode
- `best` for running tests in native mode with fallback to byte code, if native compilation is not available
- `js` for running tests in JavaScript using Node.js
- `wasm` for running tests in Wasm using Node.js

For instance:

```
(library
 (name foo)
 (inline_tests (modes byte best js wasm))
 (preprocess (pps ppx_expect)))
```

### Specifying Inline Test Dependencies

If your tests are reading files, you must tell Dune by adding a `deps` field the `inline_tests` field. The argument of this `deps` field follows the usual *Dependency Specification*. For instance:

```
(library
 (name foo)
 (inline_tests (deps data.txt))
 (preprocess (pps ppx_expect)))
```

### Specifying Inline Test arguments for Parameterised Libraries

If your library is parameterised (see *library\_parameter*), you must specify which implementation of the parameters to use with the `arguments` field. For example, suppose `foo` is a parameterised library that takes parameters `a_param` and `b_param`, you can specify the implementations of these parameters for inline tests as follows:

```
(library
 (name foo)
 (parameters a_param b_param)
 (inline_tests
 (arguments a_impl b_impl)))
```

### Passing Special Arguments to the Test Runner

Under the hood, a test executable is built by Dune. Depending on the backend used, this runner might take useful command line arguments. You can specify such flags by using a `flags` field, such as:

```
(library
  (name foo)
  (inline_tests (flags (-foo bar)))
  (preprocess (pps ppx_expect)))
```

The argument of the `flags` field follows the *Ordered Set Language*.

### Passing Special Arguments to the Test Executable

To control how the test executable is built, it's possible to customize a subset of compilation options for an executable using the `executable` field. Dune gives you this ability by simply specifying command line arguments as flags. You can specify such flags by using `flags` field. For instance:

```
(library
  (name foo)
  (inline_tests
    (flags (-foo bar)
      (executable
        (flags (-foo bar))))
    (preprocess (pps ppx_expect))))
```

The argument of the `flags` field follows the *Ordered Set Language*.

### Using Additional Libraries in the Test Runner

When tests are not part of the library code, it's possible that tests require additional libraries than the library being tested. This is the case with `qtest`, as tests are written in comments. You can specify such libraries using a `libraries` field, such as:

```
(library
  (name foo)
  (inline_tests
    (backend qtest)
    (libraries bar)))
```

### Changing the Flags of the Linking Step of the Test Runner

You can use the `link_flags` field to change the linker flags passed to `ocamlopt` when building the test runner. By default, the linking flags are `-linkall`. You probably want to keep `-linkall` as one of the new list of flags (unless you know what you are doing), forcing the linker to load your test module, since the test runner doesn't depend on anything itself. This field supports `(:include ...)` forms.

```
(library
  (name foo)
  (inline_tests
    (executable
      (link_flags -linkall -noautolink -cclib -Wl,-Bstatic -cclib -lm))
    (preprocess (pps ppx_expect)))
```

## Defining Your Own Inline Test Backend

If you are writing a test framework (or for other specific cases), you might want to define your own inline tests backend. If your framework is naturally implemented by a library or PPX rewriter that's necessary to write tests, you should define this library as a backend. Otherwise simply create an empty library with your chosen backend's name.

In order to define a library as an inline tests backend, simply add an `inline_tests.backend` field to the library stanza. An inline tests backend is specified by four parameters:

1. How to create the test runner
2. How to build the test runner
3. How to run the test runner
4. Optionally how to run the test runner to list partitions

These four parameters can be specified inside the `inline_tests.backend` field, which accepts the following fields:

```
(generate_runner    <action>)
(runner_libraries   (<ocaml-libraries>))
(flags              <flags>)
(list_partitions_flags <flags>)
(extends            (<backends>))
```

For instance:

`<action>` follows the *Actions* specification. It describes an action that should be executed in the library's directory using this backend for their tests. It's expected that the action will produce some OCaml code on its standard output. This code will constitute the test runner. The action can use the following additional variables:

- `%{library-name}` — the name of the library being tested
- `%{impl-files}` — the list of implementation files in the library, i.e., all the `.ml` and `.re` files
- `%{intf-files}` — the list of interface files in the library, i.e., all the `.mli` and `.rei` files

The `runner_libraries` field specifies what OCaml libraries the test runner uses. For instance, if the `generate_runner` actions generates something like `My_test_framework.runtests ()`, then you should probably put `my_test_framework` in the `runner_libraries` field.

If your test runner needs specific flags, you should pass them in the `flags` field. You can use the `%{library-name}` variable in this field.

If your test runner supports test partitions, you should pass the flags necessary for listing partitions in the `list_partitions_flags` field. In such scenario, the `flags` field will also accept a `%{partition}` variable.

Finally, a backend can be an extension of another backend. In this case, you must specify this in the `extends` field. For instance, `ppx_expect` is an extension of `ppx_inline_test`. It's possible to use a backend with several extensions in a library; however, there must be exactly one *root backend*, i.e., exactly one backend that isn't an extension of another one.

When using a backend with extensions, the various fields are simply concatenated. The order in which they are concatenated is unspecified; however, if a backend `b` extends a backend `a`, then `a` will always come before `b`.

## Example of Backend

In this example, we put tests in comments of the form:

```
(*TEST: assert (fact 5 = 120) *)
```

The backend for such a framework looks like this:

```
(library
  (name simple_tests)
  (inline_tests.backend
    (generate_runner (run sed "s/(\\*TEST:\\(.*\\)\\*)/let () = \\1;;/" %{impl-files}))))
```

Now all you have to do is write `(inline_tests ((backend simple_tests)))` wherever you want to write such tests. Note that this is only an example. We don't recommend using `sed` in your build, as this would cause portability problems.

### 3.13.3 Custom Tests

For the common case of running an executable as a test, write a *test stanza* in your dune file:

```
(test
  (name my_test_program))
```

This declares a test executable whose main module is `my_test_program.ml` and arranges for `dune runtest` to run it. Dune will report a test failure if the executable exits with a nonzero status.

The *tests* stanza is the multi-test form of *test*:

```
(tests (names test1 test2))
```

More generally, test actions are attached to the `runtest` alias. As a result, you can also add an action to this alias in any directory in order to define custom tests. For instance, if you have a binary `tests.exe` that you want to run as part of running your test suite, simply add this to a dune file:

```
(rule
  (alias runtest)
  (action (run ./tests.exe)))
```

### Diffing the Result

It's often the case that we want to compare the actual output of a test to an expected one. For that, Dune offers the `diff` command, which in essence is the same as running the `diff` tool, except that it's more integrated in Dune, especially with the `promote` command. For instance, let's consider this test:

```
(rule
  (with-stdout-to tests.output (run ./tests.exe)))

(rule
  (alias runtest)
  (action (diff tests.expected tests.output)))
```

After having run `tests.exe` and dumping its output to `tests.output`, Dune will compare the latter to `tests.expected`. In case of mismatch, Dune will print a diff and then the `dune promote` command can be used to copy over the generated `test.output` file to `tests.expected` in the source tree.

Alternatively, the *tests* also supports this style of tests.

```
(tests (names tests))
```

Dune expects the existence of a `tests.expected` file to infer that this is an expected test.

This provides a nice way of dealing with the usual *write code*, *run*, and *promote* cycle of testing. For instance:

```
$ dune runtest
[...]
-tests.expected
+tests.output
File "tests.expected", line 1, characters 0-1:
-Hello, world!
+Good bye!
$ dune promote
Promoting _build/default/tests.output to tests.expected.
```

Note that if available, the diffing is done using the `patdiff` tool, which displays nicer looking diffs than the standard `diff` tool. You can change that by passing `--diff-command CMD` to Dune.

### 3.13.4 Cram Tests

Cram tests are expectation tests written in a shell-like syntax. They are ideal for testing binaries. Cram tests are automatically discovered from files or directories with a `.t` extension. By default, this has been enabled since Dune 3.0. For older versions, it must be manually enabled in the `dune-project` file:

```
(lang dune 2.7)
(cram enable)
```

#### File Tests

To define a standalone test, we create a `.t` file. For example, `foo.t`:

```
Simplest possible Cram test
$ echo "testing"
```

This simple example demonstrates two components of Cram tests: comments and commands. See *Cram Tests* for a description of the syntax.

To run the test and promote the results:

```
$ dune runtest
$ dune promote
```

We now see the output of the command:

```
Simplest possible cram test
$ echo "testing"
testing
```

This is the main advantage of expect tests. We don't need to write assertions manually; instead we detect failure when the command produces a different output than what is recorded in the test script.

For example, here's an example of how we'd test the `wc` utility. `wc.t`:

```
We create a test artifact called "foo"
$ cat >foo <<EOF
> foo
> bar
> baz
> EOF
```

(continues on next page)

(continued from previous page)

After creating the fixture, we want to verify that `wc` gives us the right result:

```
$ wc -l foo | awk '{ print $1 }'  
4
```

The above example uses the doc syntax, piping the subsequent lines to `cat`. This is convenient for creating small test artifacts.

### Directory Tests

In the above example we used `cat` to create the test artifact, but what if there are too many artifacts to comfortably fit in test file? Or some of the artifacts are binary?

It's possible to include the artifacts as normal files or directories, provided the test is defined as a directory. The name of the test directory must end with `.t` and must include a `run.t` as the test script. Everything else in that directory is treated as raw data for the test. It's not possible to define rules using `dune` files in such a directory.

We convert the `wc` test above into a directory test `wc.t`:

```
$ ls wc.t  
run.t foo.txt bar/
```

This defines a directory test `wc.t` which must include a `run.t` file as the test script, with `foo.txt` and `bar` are test artifacts. We may then access their contents in the test script `run.t`:

```
Testing wc:  
$ wc -l foo | awk '{ print $1 }'  
4  
$ wc -l $(ls bar) | awk '{ print $1 }'  
1231
```

#### See also

*(cram) stanza reference*

### Testing an OCaml Program

The most common testing situation involves testing an executable that is defined in Dune. For example:

```
(executable  
 (name wc)  
 (public_name wc))
```

To use this binary in the Cram test, we should depend on the binary in the test:

```
(cram  
 (deps %{{bin:wc}}))
```

### Sandboxing

Since Cram tests often create intermediate artifacts, it's important that Cram tests are executed in a clean environment. This is why all Cram tests are sandboxed. To respect sandboxing, every test should specify dependency on any artifact that might rely on using the `deps` field.

See *Sandboxing* for details about the sandboxing mechanism.

## Test Output Sanitation

In some situations, Cram tests emit non portable or non-deterministic output. We recommend sanitising such outputs using pipes. For example, we can scrub the OCaml magic number using sed as follows:

```
$ ocamlc -config | grep "cml_magic_number:" | sed 's/Caml.*/$SPECIAL_CODE/'
cml_magic_number: $SPECIAL_CODE
```

By default, Dune will scrub some paths from the output of the tests. The default list of paths is:

- The PWD of the test will be replaced by \$TESTCASE\_ROOT
- The temporary directory for the current script will be replaced by \$TMPDIR

To add additional paths to this sanitation mechanism, it's sufficient to modify the standard `BUILD_PATH_PREFIX_MAP` environment variable. For example:

```
$ export BUILD_PATH_PREFIX_MAP="HOME=$HOME:$BUILD_PATH_PREFIX_MAP"
$ echo $HOME
$HOME
```

Note: Unlike Dune's version of Cram, the original specification for Cram supports regular expression and glob filtering for matching output. We chose not to implement this feature because it breaks the test, diff, and accept cycle. With regex or glob matching, the output must now be manually inspected and possibly updated. We consider the postprocessing approach described here as superior and will not introduce output matchers.

## 3.14 How to Bundle Resources

This guide will show you how to configure Dune to generate modules with string resources from other files in your project.

### 3.14.1 Folder Structure

```
$ tree src
src
├── lib
│   └── my_lib
│       ├── dune
│       └── resources
│           └── site.css
```

### 3.14.2 Dune Configuration

See *progn* and *with-<outputs>-to*.

```
(rule
 (with-stdout-to
  css.ml
  (progn
   (echo "let css = {|")
   (cat resources/site.css)
   (echo "|}"))))
```

### 3.14.3 Using the Bundled Resource

```
let () = Printf.printf "%s" Css.css
```

## 3.15 How to Handle External System Dependencies (depexts)

This guide shows how to declare and query external system dependencies (depexts) — system packages like C libraries or tools that OCaml packages require at build or run time.

### 3.15.1 Checking What System Packages Are Needed

#### Note

This section applies to projects using Dune's package management. If you manage dependencies with opam, use `opam list --required-by <pkg> --external` to query depexts instead.

Once you have a lock directory (created by running `dune pkg lock` or automatically when building with package management enabled), you can list all required system packages:

```
$ dune show depexts
gnupg
unzip
```

This prints the system package names from all locked packages, deduplicated and sorted. Install these packages using your system's package manager (e.g., `apt`, `brew`, `dnf`) before building.

### 3.15.2 Declaring Depexts for Your Own Packages

Declaring depexts makes your package's system dependencies visible to consumers, whether they use Dune package management or opam.

The `depexts` field is not yet supported in the `(package)` stanza of `dune-project`. To declare depexts for your own packages, add them to a `.opam.template` file:

1. Create a file named `<package-name>.opam.template` in your project root.
2. Add the `depexts` field using opam syntax:

```
depexts: [
  ["pkg-config"]
  ["libsqlite3-dev"] {os-family = "debian"}
  ["sqlite"] {os = "macos"}
]
```

3. When opam files are generated (via `dune build` with `(generate_opam_files)` in `dune-project`), the template fields are merged into the generated opam file.

If you maintain `.opam` files manually (without `(generate_opam_files)`), add the `depexts` field directly to the `.opam` file instead.

See *Packages* for more on opam file generation and templates.

### 3.15.3 Diagnosing Build Failures

#### Note

Build failure hints for depexts only appear when using Dune's package management. When using opam, depexts are typically installed automatically by `opam depext` or `opam install`.

When a locked package fails to build and has associated depexts, Dune prints a hint after the error:

```
Error: Program pkg-config not found in the tree or in PATH
(context: default)
Hint: You may want to verify the following depexts are installed:
- gnupg
- unzip
```

If you see this hint, install the listed system packages and retry the build.

#### See also

##### *How Package Management Works*

How Dune's package management works, including how depexts are resolved.

##### **opam Manual: depexts field**

Specification of the `depexts` field in opam files.

## 3.16 How to Load a Project in a Toplevel

It is possible to use OCaml code in an interactive way, by typing an expression, which gets evaluated and its result printed. Such a program is called a *toplevel*, or REPL (Read-Eval-Print Loop).

The compiler distribution comes with a small REPL called simply `ocaml`, and the community has developed enhanced versions such as `UTop`.

### 3.16.1 Building a Specialized UTop Executable

It is possible to generate a specialized version of `UTop` that embeds the current project. To do so, use the following command:

```
$ dune utop
```

The interactive session will start with all the modules loaded.

If some of the libraries are PPX rewriters, the phrases you type in the toplevel will be rewritten with these PPX rewriters. Similarly, PPX drivers defined in the project will be available.

### 3.16.2 Loading the Project in a Toplevel

It is also possible to load Dune projects in any toplevel. To do that, simply execute the following in your toplevel:

```
# #use_output "dune ocaml top";;
```

`dune ocaml top` is a Dune command that builds all the libraries in the current directory and subdirectories and outputs the relevant toplevel directives (`#directory` and `#load`) to make the various modules available in the toplevel.

### 3.16.3 Loading a Single Module in a Toplevel

It's also possible to load individual modules for interactive development. Use the following dune command:

```
# #use_output "dune ocaml top-module foo.ml";;
```

This will print directives that will load `foo.ml` without sealing it behind `foo.mli`. This is particularly useful for peeking and prodding at a module's internals.

## 3.17 Using Rule Generation

Sometimes it can be useful to generate Dune rules that depend on the file system layout or on the content of configuration files. This often happens for integration tests.

In this document, we will see two ways to encode this behavior.

We suppose that we are testing an executable named `tool`. There are some input files named `*.input`, output files named `*.output`, and we want to ensure that when running `tool` on `x.input`, the standard output corresponds to `x.output`.

### 3.17.1 The Generate-Include-Commit Pattern

#### Note

This is the most common way to do this. It has a couple drawbacks listed below, but you should start with this pattern.

What we are going to do is:

- generate a `dune.inc` file;
- include it in our main dune file;
- commit the generated code in the source repository.

This creates a loop: a program (the “generator”) looks at the file system and creates a `dune.inc` file. Changes to the file system (for example, if a test is added) mean that a change in `dune.inc` will be promoted. These generated rules are included in the main dune file, so `dune runtest` will run the tests. Finally, the generated file is part of the source repository, so it is not necessary to run several commands to run the test suite.

Let's expand a bit on how to achieve this.

#### Generating a `dune.inc` File

Create a `gen` subdirectory and create a `gen.ml` file in it:

`gen/gen.ml`

```
let generate_rules base =
  Printf.printf
    { |
      (rule
        (with-stdout-to %s.gen
          (run %#{bin:tool} %s.input)))

      (rule
```

(continues on next page)

(continued from previous page)

```

    (alias runtest)
    (action
      (diff %s.output %s.gen)))
  }
  base base base base

let () =
  Sys.readdir "."
  |> Array.to_list
  |> List.sort String.compare
  |> List.filter_map (Filename.chop_suffix_opt ~suffix:".input")
  |> List.iter generate_rules

```

Create a dune file in that directory:

#### gen/dune

```

(executable
 (name gen))

```

This defines an executable that lists `*.input` files in the current directory and outputs rules on its standard output.

#### **Note**

It is important to sort the input files to ensure that the output is independent from the order in which `Sys.readdir` returns the files.

For each input file, we output two rules:

- The first one creates a `x.gen` file that corresponds to the actual output.
- The second uses a *diff* action to compare the actual output to the expected output. If it is different, `dune runtest` will display the difference, which can be accepted by `dune promote`.

#### **Note**

It is possible to have more complicated logic here. For example, to pass different arguments to `tool` depending on the presence of a `*.args` file. To do that, check if `*.args` exists in `generate_rules` and emit a different `(run ...)` action.

### Including it in the Main dune File

Our main test dune file contains the following:

#### dune

```

(include dune.inc)

(rule
 (deps (source_tree .))
 (with-stdout-to
  dune.inc.gen
  (run gen/gen.exe)))

```

(continues on next page)

(continued from previous page)

```
(rule
  (alias runtest)
  (action
    (diff dune.inc dune.inc.gen)))
```

In addition to including the contents of `dune.inc`, we use the same pattern as before: `dune.inc.gen` is the actual output of the generator, and `dune.inc` is the expected output. At runtime, the generator will read the contents of the current directory (where the `*.input` and `*.output` files are located), so we record `(source_tree .)` as a dependency to make it run again if a file is created, for example.

### Commit the Generated Code In The Source Repository

To make this work, we have a final step to do. We have to add the generated file to our source tree. But since it is generated, we will have to first create an empty file, run the test, and promote the result.

```
$ touch dune.inc
$ dune runtest
+ (rule
+   (with-stdout-to a.gen
+     (run %{bin:tool} a.input)))
+
+ (rule
+   (alias runtest)
+   (action
+     (diff a.output a.gen)))
$ dune promote dune.inc
$ git add dune.inc
```

Now, running `dune runtest` will run the test suite.

### Notes

This pattern is “correct”: it will execute all tests and make sure the list of tests is up to date. But when adding a test, it is necessary to first run `dune runtest`, promote the result, and then re-run `dune runtest` to actually run the test (and possibly promote the result of the test itself).

There is a variant of this pattern which will promote the output automatically instead of using a manual promotion step. This variant can be used either for the test list or for the individual tests.

To use it in the test list, replace the `dune` file by this version:

#### **dune (alternative version)**

```
(include dune.inc)

(rule
  (mode promote)
  (alias runtest)
  (deps (source_tree .))
  (with-stdout-to
    dune.inc
    (run gen/gen.exe)))
```

Using this version, `dune runtest` will directly replace `dune.inc` with an updated version.

Another caveat of this approach is that the generator needs to emit the same output on all systems. For example, if some tests should be skipped on Linux, the generator can not just filter the corresponding tests depending on `Sys.os_type`. It has to consistently emit a `(enabled_if)` field for the rules.

### 3.17.2 Using `(dynamic_include)`

Added in version 3.14.

This technique relies on `dynamic_include`, which is more flexible than `include`. The difference is that the intermediate `dune.inc` file does not need to be part of the source tree. It will only be generated by a rule and be present in the `_build` directory.

At first it looks like it would be possible to reuse the same pattern as above: change `include` to `dynamic_include` and delete the `dune.inc` file. However, it is not possible. The reason is that rules are loaded per directory, and there needs to be a strict order (no cycles) between directories for this to work.

So, instead we are going to:

- generate `dune.inc` in a subdirectory named `generate`, and
- include these rules in a subdirectory named `run`.

These subdirectories do not need to be actual directories. They can be emulated through `subdir`.

To do this, we can create the following `dune` file in the same directory as the `*.input` and `*.output` files.

**dune**

```
(executable
 (name gen))

(subdir run
 (dynamic_include ../generate/dune.inc))

(subdir generate
 (rule
 (deps (glob_files ../*.input))
 (action
 (with-stdout-to dune.inc
 (run ../gen.exe))))))
```

Then create the following `gen.ml` file. Note that here we can define it in the same directory.

**gen.ml**

```
let generate_rules base =
  Printf.printf
  { |
    (rule
     (with-stdout-to %s.gen
      (run %{bin:tool} ../%s.input)))

    (rule
     (alias runtest)
     (action
      (diff ../%s.output %s.gen)))
    |}
  base base base base
```

(continues on next page)

```
let () =
  Sys.readdir ".." |> Array.to_list |> List.sort String.compare
  |> List.filter_map (Filename.chop_suffix_opt ~suffix:".input")
  |> List.iter generate_rules
```

There are a few differences from the generator above because this one is going to be invoked from subdirectories, so it is necessary to refer to the `..` directory both in the input (which files to read) and in the output (how the rules are executed).

These two files are enough. `dune runtest` is going to generate the rules and interpret them in a single command.

## Notes

This approach is shorter, but it might be more difficult to debug because changes to the generated rules will not be visible. Also, it works in that case, but it is not possible to generate all kinds of stanzas with that pattern. See *dynamic\_include* for more information about the limitations.

## 3.18 How to Override the Default C Entrypoint With C Stubs

In some cases, it may be necessary to override the default C entry point of an OCaml program. For example, this is the case if you want to let your program handle argument wildcards expansion on Windows.

Let's consider a trivial "Hello world" program contained in a `hello.ml` file:

```
let () = print_endline "Hello, world!"
```

The default C entry point is a `main` function, originally defined in `runtime/main.c`. It can be overridden by defining a `main` function that will at some point call the OCaml runtime. Let's write such a minimal example in a `main.c` file:

```
#include <stdio.h>

#define CAML_INTERNALS
#include "caml/misc.h"
#include "caml/mlvalues.h"
#include "caml/sys.h"
#include "caml/callback.h"

/* This is the new entry point */
int main(int argc, char_os **argv)
{
  /* Here, we just print a statement */
  printf("Doing stuff before calling the OCaml runtime\n");

  /* Before calling the OCaml runtime */
  caml_main(argv);
  caml_do_exit(0);
  return 0;
}
```

The `foreign_stubs` stanza can be leveraged to compile and link our OCaml program with the new C entry point defined in `main.c`:

```
(executable
 (name hello)
 (foreign_stubs
 (language c)
 (names main)))
```

With this dune file, the whole program can be compiled by merely calling `dune build`. When run, the output shows that it calls the custom entry point we defined:

```
$ dune build
$ _build/default/hello.exe
Doing stuff before calling the OCaml runtime
Hello, world!
```

## 3.19 How to Organize Multiple Executables in Different Directories

A common question when a project grows beyond a single program is: “I want to build several executables, each living in its own directory. How do I set that up?”

Dune does not impose a particular layout on your project. You are free to place your executables in whatever directories make sense to you. The key idea is that you *describe* each directory with a dune file, and Dune assembles the whole project from those descriptions.

### 3.19.1 The Mental Model

Three rules are enough to organize almost any project:

- A project has a single `dune-project` file at its root. It records project-wide information, most importantly the version of the Dune language in use.
- Every directory that contains something to build has its own dune file describing what is in that directory.
- An executable is defined by an *executable* stanza in the dune file of the directory where its sources live.

So to have executables in several directories, you simply put a dune file with an `executable` stanza in each of those directories. Nothing forces them to sit together, and there is no central list of executables to maintain.

Consider a project with a `server` and a `client`, each in its own directory:

```
myproject/
├── dune-project
├── client/
│   ├── dune
│   └── client.ml
└── server/
    ├── dune
    └── server.ml
```

### 3.19.2 The `dune-project` File

There is exactly one `dune-project` file, and it lives at the root of the project. The directories below it do not need their own. At a minimum it declares the language version:

```
(lang dune 3.25)
```

### 3.19.3 Executables in Multiple Subdirectories

Each executable is described where its sources are. The `client/dune` file declares the client:

```
(executable
 (name client))
```

And `server/dune` declares the server in the same way:

```
(executable
 (name server))
```

With the source files in place, for example `client/client.ml`:

```
let () = print_endline "client starting"
```

Dune builds both executables from the root of the project:

```
$ dune build
```

You run each one by giving its path relative to the project root, using the `.exe` extension regardless of platform:

```
$ dune exec client/client.exe
client starting

$ dune exec server/server.exe
server starting
```

The directory names (`client` and `server`) are entirely your choice, and the directories can be nested as deeply as you like. Dune discovers them by reading the `dune` files it finds while traversing the tree.

### 3.19.4 Sharing Code Between Executables

As soon as two executables need to share code, move that code into a library. A library is just another directory with a `dune` file, this time containing a *library* stanza. Place it wherever you like; a `lib/` directory is a common convention:

```
myproject/
├── dune-project
├── client/
│   ├── dune
│   └── client.ml
├── server/
│   ├── dune
│   └── server.ml
└── lib/
    └── protocol/
        ├── dune
        └── protocol.ml
```

The `lib/protocol/dune` file defines the shared library:

```
(library
 (name protocol))
```

Its modules are written in that directory, for example `lib/protocol/protocol.ml`:

```
let greeting = "hello from protocol"
```

Each executable that wants the library lists it under `libraries`. Note that a library is referred to by the name from its stanza, not by its directory path, so it does not matter where in the tree the library lives. The updated `client/dune` becomes:

```
(executable
 (name client)
 (libraries protocol))
```

The client can now use the library's modules directly:

```
let () = print_endline Protocol.greeting
```

The same `(libraries protocol)` line added to `server/dune` gives the server access to the very same code. See *Library Dependencies* for the full syntax of the `libraries` field.

### 3.19.5 A Note on `dune init`

The `dune init` command can scaffold projects, executables, and libraries for you, which is handy when you are starting out. It is only a convenience, though: it writes the same `dune-project` and `dune` files shown above. Once you understand the layout, you can create and arrange these files by hand in whatever structure suits your project.

## 3.20 How to Release a Binary Distribution of an Application on GitHub with Dune

This guide will show you how to write a GitHub Action which builds a binary distribution of an application for various platforms using Dune package management, and uploads the compiled artifacts to a GitHub release.

We'll make a workflow called "Release" which will run every time a tag is pushed to the repo on GitHub. The workflow will build the project for the targets `x86_64-linux`, `x86_64-macos`, and `aarch64-macos`, and then upload a gzipped tarball of the built artifacts to a GitHub release named after the tag.

Dune will be installed by the `setup-dune` action.

The following is the manifest for a GitHub Action workflow which releases a binary distribution of a package named `my_app`.

```
name: Release

on:
  push:
    tags:
      - '*'

jobs:
  release-unix:
    name: Release for ${matrix.name}
    runs-on: ${matrix.os}
    strategy:
      matrix:
        include:
          - os: macos-15-intel
            name: x86_64-macos
```

(continues on next page)

```

- os: macos-15
  name: aarch64-macos
- os: ubuntu-latest
  name: x86_64-linux
steps:
- uses: actions/checkout@v7

- name: Install Dune
  uses: ocaml-dune/setup-dune@v2

- name: Build the project
  run: dune build @install --release --only-packages my_app

- name: Set environment variable to the archive name minus the file extension
  run: echo OUT_NAME=my_app-${{ github.ref_name }}-${{ matrix.name }} >> $GITHUB_
↪ENV

- name: Release a tarball of build outputs
  run: |
    mkdir -p "$OUT_NAME"
    cp -rlf _build/install/default/* "$OUT_NAME"
    tar czf "$OUT_NAME.tar.gz" "$OUT_NAME"

- name: Upload artifacts
  uses: ncipollo/release-action@v1
  with:
    allowUpdates: true
    artifacts: "/*.tar.gz"

```

With the above manifest in a file `.github/workflows/release.yml`, whenever a tag is pushed to the repo, gzipped tarballs will be uploaded to a new release of the project named after the tag, in files named `my_app-<TAG>-x86_64-linux.tar.gz`, `my_app-<TAG>-x86_64-macos.tar.gz`, and `my_app-<TAG>-aarch64-macos.tar.gz`.

## 3.21 How to make a Homebrew Package with Dune

This guide will show you how to make a Homebrew package for an application using Dune package management. The only dependency of the Homebrew package will be Dune, and all OCaml dependencies (including the OCaml compiler) will be installed by Dune while building the Homebrew package.

To use Dune package management to build a project as a Homebrew package, the project must have a source archive hosted online somewhere (e.g. a gzipped tarball on the project's GitHub release page).

Before making a Homebrew package, it's a good idea to familiarize yourself with Homebrew's terminology and packaging conventions [here](#).

Homebrew packages are recommended to be source-based, and for the source code to be explicitly versioned, so for this example assume `my_app` has a versioned archive hosted on GitHub with version `0.1.0`.

Homebrew can generate a starting point for a formula if you point it at a source archive hosted on GitHub:

```
$ brew create https://github.com/me/my_app/archive/refs/tags/0.1.0.tar.gz
```

A source archive like the one in the above command is generated when you release a project on GitHub. The above command will generate a file named `my_app.rb` in your current tap. All the project metadata will be filled in automatically based on the project on GitHub. All we need to do now is to specify dependencies and the commands `brew` should run when installing the package.

Here's the complete formula for `my_app`. Note that the `test` section is intended to be a sanity check of the core functionality of the package, not a complete integration test suite. Read more about Homebrew package tests [here](#). Note however that tests run in an environment without access to build dependencies such as Dune, so `dune runtest` can't be used to test Homebrew packages.

```
class MyApp < Formula
  desc "My awesome app"
  homepage "https://github.com/me/my_app"
  url "https://github.com/me/my_app/releases/download/0.1.0/0.1.0.tar.gz"
  sha256 "eb8705de406441675747a639351d0d59bffe7b9f5b05ec9b6e11b4c4c9d7a6ee"
  license "MIT"

  depends_on "dune" => :build

  def install
    # Uncomment if the source archive lacks a lockdir:
    # system "dune", "pkg", "lock"
    system "dune", "build", "@install", "--release", "--only-packages", "my_app"
    system "dune", "install", "--prefix=#{prefix}", "my_app"
  end

  test do
    # Test your application here!
    system "my_app", "--version"
  end
end
```

This assumes that the name of the package in the source archive is `my_app`. That is, the archive contains a `dune-project` file defining a package named `my_app`, or that the archive contains a `my_app.opam`. The archive may contain multiple packages provided that `my_app` is one of them.

Note the comment at the beginning of the `install` method. The commented-out code invokes Dune's solver to compute the transitive closure of packages that will be built as dependencies of `my_app`, and stores the result in a directory named `dune.lock` - a "Lock Directory" or "lockdir" for short. Solving dependencies in the `install` method should be avoided when possible. This is because Dune solves dependencies in the context of the current tip of the [Opam Repository](#), which changes as new Opam packages are released. This means that the exact solution computed by Dune can change as new versions of dependencies come out. Homebrew encourages package builds to be [reproducible](#) when possible, but solving dependencies each time a package is installed prevents that package from being built reproducibly. To allow reproducible builds, always include the project's `lockdir` in its source archive (generate a `lockdir` by running `dune pkg lock`) when releasing a package. Only solve dependencies in the `install` method when packaging a project whose source archive lacks a `lockdir`.

If there are any packages with external dependencies (i.e. `depexts`) in `my_app`'s transitive dependency closure, their corresponding Homebrew package must be added as a dependency of `my_app`'s Homebrew packages by adding a `depends_on` entry for each. List all the external dependencies among `my_app`'s transitive dependency closure by running:

```
$ dune show depexts
```

External dependencies can be platform-specific, so if you're planning to make the Homebrew package available for macOS, be sure to run the above command on a Mac to determine which external dependencies need to be added to

the formula.

### 3.22 How to Use Opam Alongside Dune Package Management

This guide will show you a workflow for gradually adopting Dune Package Management, while retaining the use of an Opam switch as the default way of resolving dependencies of a project. It assumes you have a project with dependencies, and are managing those dependencies with an Opam switch. By the end of this guide you'll have enabled Dune Package Management for your project, however by default Dune will still use an Opam switch to resolve your project's dependencies. By setting an environment variable or passing an extra argument to commands like `dune build` you'll be able to toggle between an Opam-based workflow and a purely Dune-based workflow.

Create a file named `dune-workspace` at the root of your project, with the following contents:

```
(lang dune 3.21)

(pkg disabled)
```

Since `dune-workspace` is the default workspace file, this tells Dune that by default, don't use its internal package management mechanism to resolve the project's dependencies, even in the presence of a lockdir.

Now create a second file in the project root named `dune-workspace.pkg`. The name of this file isn't important, but conventionally alternative workspace files begin with the `dune-workspace.` prefix. Add the following to this new file:

```
(lang dune 3.21)

(pkg enabled)
```

With these two files in place, by default Dune will use the specified Opam switch to resolve your project's dependencies. However, setting the environment variable `DUNE_WORKSPACE=dune-workspace.pkg` will cause Dune to use its internal package management mechanism to resolve dependencies instead. Unset the `DUNE_WORKSPACE` environment variable to return to an Opam-based workflow.

Alternatively to setting the `DUNE_WORKSPACE` environment variable, invoking commands like `dune build`, `dune exec`, and `dune pkg lock` with the argument `--workspace=dune-workspace.pkg` also has the effect of changing which workspace file Dune will use to the one which enables package management.

### 3.23 How to Configure Editors with Package Management

This document aims to help people set up their editors for use with Dune package management. Different configurations are required than when using opam to manage dependencies.

#### Note

This list is not exhaustive. If you have a configuration for your editor that is not included here, we would welcome a [pull request to the Dune repository](#) documenting the configuration.

#### 3.23.1 Emacs

For Emacs, we recommend using `ocaml-eglot` mode with Dune package management. Follow the [instructions in the OCaml-eglot repository](#).

### 3.23.2 Neovim

The instructions on how to configure `ocaml.nvim`, the OCaml plugin for Neovim can be found in the `ocaml.nvim` repository.

### 3.23.3 Visual Studio Code

The configuration for the VSCode OCaml Platform extension is described in the `vscode-ocaml-platform` repository.

### 3.23.4 More information

For general information on editor set up for OCaml (not including configuration specific to dune package management) check out the [community documentation on OCaml.org](#).

## 3.24 How to Set Up Shell Command Completion

Shell command completion refers to a common feature in various shells: hitting `<TAB>` after a partially-typed command will print matching suggestions. This can apply to command names, flags, arguments...

Dune offers completion for commands, subcommands, and flags, for Bash, Z shell, and PowerShell. The various shells and systems require specific configuration setups, and this document aims to help users get completion running.

### 3.24.1 Bash

Command completion for bash might require `bash-completion`. You will then need to create the bash completion script (by running `dune completion bash`) and write that script to an appropriate place: by default that should be `~/.local/share/bash-completion/completions/dune` for a user-local configuration, or `/usr/local/share/bash-completion/completions/dune` for a system-wide installation. If you use `opam` switches and wish to have switch-local completion, add the script to `<switch prefix>/share/bash-completion/completions/dune`.

A different option would be to add to your `.bashrc` the following line: `eval "$(dune completion bash)"`.

You might need to restart your shell to see the effects.

### 3.24.2 Zsh

Command completion for zsh might require `zsh-completions`. You will then need to create the zsh completion script (by running `dune completion zsh`) and write that script to a place that will get autoloaded: directories present in your `$fpath` (by default it should be `~/.local/share/zsh/site-functions`). If you wish to create a dedicated completion directory and add it to your `$fpath`, be sure to add the following in your `.zshrc` but **before** any call to `compinit`: `fpath+=<your dir>`.

You might need to restart your shell to see the effects.

### 3.24.3 PowerShell

Command completion for PowerShell should work out of the box: simply run `dune completion powershell >> $PROFILE.CurrentUserCurrentHost`. Alternatively, add the following to your profile script: `dune completion powershell | Out-String | Invoke-Expression`.

You might need to restart your shell to see the effects.

## 3.25 How to set up Continuous Integration with Package Management

For many projects published it can be useful to set up a [continuous integration](#) (CI) system. This allows checking whether the code builds on other computers, the tests pass and helps to assess contributions from other people.

There are numerous CI systems available. In this document we will be using GitHub Actions, as it is integrated in the commonly used GitHub platform and free to use.

### 3.25.1 What are Actions

Projects on GitHub can set up [Actions](#), that is workflows defined in YAML files. These definitions are placed in the `.github/workflows/` folder in the Git repository.

These workflows consist of the execution of a sequence of actions, like running shell commands. It is also possible to use pre-defined actions or even user-submitted functionality.

### 3.25.2 `setup-dune`

`setup-dune` is one such custom action maintained by the OCaml community to help with setting up builds with Dune.

It installs Dune in the GitHub runner environment and will attempt to build the source code using Dune package management.

`.github/workflow/ci.yaml`

```
name: Build and test

on: [push, pull_request]

jobs:
  build:
    strategy:
      matrix:
        runs-on: [ ubuntu-latest, macos-latest ]
    runs-on: ${ matrix.runs-on }

    steps:
      - name: Checkout code
        uses: actions/checkout@v7
      - name: Build project with dune
        uses: ocaml-dune/setup-dune@v2
```

We define a job called `build`, which will be run on Ubuntu and macOS (a [list of possible options](#) can be found in the GitHub documentation).

This job first checks out the projects' source code using an action maintained by GitHub and then uses the `setup-dune` action to install Dune on the system and build the project.

`setup-dune` will enable package management by default and attempt to build the project with the current nightly version of Dune.

Additional configuration options can be found in the [documentation of `setup-dune`](#), which includes picking a fixed version of Dune, picking a specific OCaml compiler etc.

This configuration file has to be added to the repository and pushed to GitHub. Some configuration to enable actions in the repository might be required, check out the [GitHub documentation on repository settings](#) related to actions.

Once set up, this action will be run on the commit of every push to every branch as well as on every pull request. The results of running the actions are visible as a colored dot next to the relevant commits that were processed by CI.

### ➔ See also

[GitHub Actions documentation](#) for information about the configuration format and usual workflows.

[GitHub Actions marketplace](#) for a list of published actions.

## 3.26 How to Customize Lock Directories of Developer Tools

The package management support in Dune has a feature for installing developer tools. These tools are useful for working on the code of the project but not necessary for the deployment of the code. Examples of such tools include `utop`, the OCaml LSP server or `ocamlformat`.

In general these tools do not require any specific configuration: Dune will create a lock dir implicitly and install a compatible version of the tool depending on its availability in `opam-repository`.

### 📘 Note

The configuration of lock directories of developer tools is in an *experimental* state. It is possible that before stabilizing the configuration format might be changed. This document solely documents the current state of the lock configuration with no guarantees about future compatibility.

In case you encounter issues please [open an issue](#) in the Dune issue tracker.

However, in some cases it might be necessary to override the configuration of the developer tool. This might for example be if a special fork of the tool is to be used, if a specific version should be picked or if additional constraints are required.

Configuring the lock directory for a developer tool works in the same way as configuring any other lock directory, via the `lock_dir stanza`. The difference however is, that the lock dir path for a lock dir cannot be chosen freely and must match the internal path that Dune will pick for the lock directory of said developer tool.

The format of the lock dir path is `<build-dir>/dev-tools.locks/<name-of-opam-package>`. By default the `<build-dir>` will be `_build`, unless it is configured otherwise.

### 3.26.1 A trivial example

Thus the potentially simplest configuration for a customized config for `ocamlformat` is this definition in `dune-workspace`:

```
(lang dune 3.21)
(pkg enabled)

(lock_dir
 (path "_build/dev-tools.locks/ocamlformat"))
```

The `lock_dir` stanza defines that the configuration to be set is `_build/dev-tools.locks/ocamlformat`, thus the lock directory location that Dune will pick when attempting to create a lock directory and run the solver for `ocamlformat`.

However, given the lock dir does not contain any additional definitions this does not customize anything.

### 3.26.2 Adding more customization

Thus a more complete example which exercises a lot of useful options to configure a custom ocamlformat can look like this `dune-workspace` configuration:

```
(lang dune 3.21)
(pkg enabled)

(repository
 (name oxcaml)
 (url "git+https://github.com/oxcaml/opam-repository"))

(pin
 (name ocamlbuild)
 (url "git+https://github.com/Sudha247/ocamlbuild#oxcaml+dune")
 (package
  (name ocamlbuild)
  (version 0.15.0+ox)))

(lock_dir
 (path "_build/.dev-tools.locks/ocamlformat")
 (pins ocamlbuild)
 (constraints
  (ocaml-lsp-server (= 1.19.0+ox))
  (ocaml-variants (= 5.2.0+ox)))
 (repositories overlay oxcaml upstream))
```

It specifies that the lock tool should use a pinned version of `ocamlbuild` as well as custom repositories in a specific order (adding the `oxcaml` repository in between the repositories that are defined out of the box). Constraints are added on the versions of other packages to be selected for building `ocamlformat`.

#### See also

`lock_dir`. Reference for the `lock_dir` stanza, documenting all possible customization options.

## 3.27 How to Use Multiple Build Contexts

Build contexts let Dune compile the same sources in multiple isolated environments. This is useful when one build is part of the normal workflow and another is only needed on demand, such as a `ThreadSanitizer` build.

If you have used Dune before, you have been using one context already: the `default` context. You can see this in the build directory at `_build/default`, which is where Dune will put the build artifacts of your build. By adding more contexts, you can ask Dune to build the same sources in different environments while keeping the artifacts separate.

This can be useful in certain cases: for example you want to test your project with multiple compilers without having to edit your `dune-project` file, or `opam` switches every time. We will see a use case of this with **ThreadSanitizer** below.

#### Note

`ThreadSanitizer` is a dynamic data race detector that requires a special version of the OCaml compiler. You can find more information at the [manual page](#). However, familiarity with `ThreadSanitizer` is not necessary as this is only an example to show how Dune contexts can be used.

This guide shows two ways to do that:

- with Dune package management and a separate `lock_dir`
- with an `opam` switch for the alternate compiler

### 3.27.1 Example Program

The examples below use a small program with a data race.

Listing 1: dune-project

```
(lang dune 3.21)

(package
  (name tsan-check)
  (depends
    (ocaml
      (= 5.3.0))))
```

Listing 2: dune

```
(executable
  (name tsan_check)
  (modes native))
```

Listing 3: tsan\_check.ml

```
let x = ref 0
let n = 5_000_000

let inc () =
  for _ = 1 to n do
    x := !x + 1
  done
;;

let () =
  let d = Domain.spawn inc in
  inc ();
  Domain.join d;
  let expected = 2 * n in
  Printf.printf "x=%d expected=%d\n%!" !x expected;
  assert (!x = expected)
;;
```

Build and run it with your current compiler:

```
$ dune exec ./tsan_check.exe
```

With a stock compiler, you may see the assertion fail, but you will not get a ThreadSanitizer report explaining the race.

### 3.27.2 Use Dune Package Management

First, declare `ocaml-option-tsan` as an optional dependency in `dune-workspace`, for the ThreadSanitizer build. Then create a `dune-workspace` file that defines two lock directories and attaches one context to each:

Listing 4: `dune-workspace`

```
(lang dune 3.21)

(lock_dir
 (path dune.lock))

(lock_dir
 (path dune-tsan.lock)
 (depopts ocaml-option-tsan))

(context default)

(context
 (default
  (name tsan)
  (lock_dir dune-tsan.lock)))
```

Generate the lock directories:

```
$ dune pkg lock dune.lock dune-tsan.lock
```

Build and run the binaries:

```
$ dune exec --context=default ./tsan_check.exe
$ dune exec --context=tsan ./tsan_check.exe
```

The `tsan` binary reports the data race, while the `default` binary uses your normal compiler setup.

#### ThreadSanitizer output

```
WARNING: ThreadSanitizer: data race (pid=1662846)
Write of size 8 at 0x7fa28ecffa58 by main thread (mutexes: write M78):
 #0 camlDune__exe__Tsan_check.inc_272 <null> (tsan_check.exe+0x483e2)
 #1 camlDune__exe__Tsan_check.entry <null> (tsan_check.exe+0x484df)
 #2 caml_program <null> (tsan_check.exe+0x460a9)
 #3 caml_start_program <null> (tsan_check.exe+0xdca47)
 #4 caml_startup_common runtime/startup_nat.c:132 (tsan_check.exe+0xdc1f0)
 #5 caml_startup_common runtime/startup_nat.c:88 (tsan_check.exe+0xdc1f0)
 #6 caml_startup_exn runtime/startup_nat.c:139 (tsan_check.exe+0xdc29b)
 #7 caml_startup runtime/startup_nat.c:144 (tsan_check.exe+0xdc29b)
 #8 caml_main runtime/startup_nat.c:151 (tsan_check.exe+0xdc29b)
 #9 main runtime/main.c:37 (tsan_check.exe+0x45bf9)

Previous write of size 8 at 0x7fa28ecffa58 by thread T1 (mutexes: write M83):
 #0 camlDune__exe__Tsan_check.inc_272 <null> (tsan_check.exe+0x483e2)
 #1 camlStdlib__Domain.body_735 <null> (tsan_check.exe+0x7b8ef)
 #2 caml_start_program <null> (tsan_check.exe+0xdca47)
 #3 caml_callback_exn runtime/callback.c:201 (tsan_check.exe+0x9ee53)
 #4 domain_thread_func runtime/domain.c:1215 (tsan_check.exe+0xa32d6)
```

(continues on next page)

(continued from previous page)

```

Mutex M78 (0x561082d9e728) created at:
  #0 pthread_mutex_init ../../../../src/libsanitizer/tsan/tsan_interceptors_posix.
↳cpp:1227 (libtsan.so.0+0x4bee1)
  #1 caml_plat_mutex_init runtime/platform.c:57 (tsan_check.exe+0xc9e6a)
  #2 caml_init_domains runtime/domain.c:943 (tsan_check.exe+0xa301e)
  #3 caml_init_gc runtime/gc_ctrl.c:353 (tsan_check.exe+0xaff83)
  #4 caml_startup_common runtime/startup_nat.c:111 (tsan_check.exe+0xdc0d7)
  #5 caml_startup_common runtime/startup_nat.c:88 (tsan_check.exe+0xdc0d7)
  #6 caml_startup_exn runtime/startup_nat.c:139 (tsan_check.exe+0xdc29b)
  #7 caml_startup runtime/startup_nat.c:144 (tsan_check.exe+0xdc29b)
  #8 caml_main runtime/startup_nat.c:151 (tsan_check.exe+0xdc29b)
  #9 main runtime/main.c:37 (tsan_check.exe+0x45bf9)

Mutex M83 (0x561082d9e840) created at:
  #0 pthread_mutex_init ../../../../src/libsanitizer/tsan/tsan_interceptors_posix.
↳cpp:1227 (libtsan.so.0+0x4bee1)
  #1 caml_plat_mutex_init runtime/platform.c:57 (tsan_check.exe+0xc9e6a)
  #2 caml_init_domains runtime/domain.c:943 (tsan_check.exe+0xa301e)
  #3 caml_init_gc runtime/gc_ctrl.c:353 (tsan_check.exe+0xaff83)
  #4 caml_startup_common runtime/startup_nat.c:111 (tsan_check.exe+0xdc0d7)
  #5 caml_startup_common runtime/startup_nat.c:88 (tsan_check.exe+0xdc0d7)
  #6 caml_startup_exn runtime/startup_nat.c:139 (tsan_check.exe+0xdc29b)
  #7 caml_startup runtime/startup_nat.c:144 (tsan_check.exe+0xdc29b)
  #8 caml_main runtime/startup_nat.c:151 (tsan_check.exe+0xdc29b)
  #9 main runtime/main.c:37 (tsan_check.exe+0x45bf9)

Thread T1 (tid=1662848, running) created by main thread at:
  #0 pthread_create ../../../../src/libsanitizer/tsan/tsan_interceptors_posix.cpp:969_
↳(libtsan.so.0+0x605b8)
  #1 caml_domain_spawn runtime/domain.c:1265 (tsan_check.exe+0xa499a)
  #2 caml_c_call <null> (tsan_check.exe+0xdc92b)
  #3 camlStdlib__Domain.spawn_730 <null> (tsan_check.exe+0x7b806)
  #4 camlDune__exe__Tsan_check.entry <null> (tsan_check.exe+0x484d1)
  #5 caml_program <null> (tsan_check.exe+0x460a9)
  #6 caml_start_program <null> (tsan_check.exe+0xdca47)
  #7 caml_startup_common runtime/startup_nat.c:132 (tsan_check.exe+0xdc1f0)
  #8 caml_startup_common runtime/startup_nat.c:88 (tsan_check.exe+0xdc1f0)
  #9 caml_startup_exn runtime/startup_nat.c:139 (tsan_check.exe+0xdc29b)
  #10 caml_startup runtime/startup_nat.c:144 (tsan_check.exe+0xdc29b)
  #11 caml_main runtime/startup_nat.c:151 (tsan_check.exe+0xdc29b)
  #12 main runtime/main.c:37 (tsan_check.exe+0x45bf9)

SUMMARY: ThreadSanitizer: data race (/path/to/directory/hello-tsan/_build/tsan/tsan_
↳check.exe+0x483e2) in camlDune__exe__Tsan_check.inc_272
=====
x=5014663 expected=10000000
Fatal error: exception Assert_failure("tsan_check.ml", 19, 2)
ThreadSanitizer: reported 1 warnings

```

### 3.27.3 Use an opam Context

The second approach uses separate opam switches instead of separate lock directories.

#### ❗ Important

This guide assumes that your ThreadSanitizer compiler environment already exists, you can find instructions in the [manual page](#). You may also want to install depexts listed in `dune show depexts`.

You can setup a ThreadSanitizer switch with the following command:

```
$ opam switch create 5.3.0+tsan ocaml-option-tsan
```

Create a dune-workspace file:

Listing 5: dune-workspace

```
(lang dune 3.21)

(context default)

(context
  (opam
    (switch 5.3.0+tsan)
    ; needs a ThreadSanitizer enabled switch like 5.3.0+tsan
    (name tsan)))
```

This keeps the current environment as the `default` context and adds a second context named `tsan`.

Check the contexts:

```
$ dune describe contexts
default
tsan
```

Run the stock binary:

```
$ dune exec --context=default ./tsan_check.exe
```

Run the ThreadSanitizer binary:

```
$ dune exec --context=tsan ./tsan_check.exe
```

This lets you keep your regular compiler in the `default` context while using an alternate opam switch for the ThreadSanitizer build.

## REFERENCE MANUAL

These documents specify the various features and languages present in Dune.

### 4.1 dune

dune files are the main part of Dune. Each dune file belongs to a Dune project (see *Dune Projects and Workspaces*). They are used to describe libraries, executables, tests, and everything else Dune needs to know about.

The syntax of dune files is described in *Lexical Conventions*.

dune files are composed of stanzas, as shown below:

```
(library
  (name mylib)
  (libraries base lwt))

(rule
  (target foo.ml)
  (deps generator/gen.exe)
  (action (run %{deps} -o %{target})))
```

The following pages describe the available stanzas and their meanings.

#### 4.1.1 executable

The executable stanza must be used to describe an executable. The format of executable stanzas is as follows:

```
(executable
  (name <name>)
  <optional-fields>)
```

<name> is a module name that contains the executable's main entry point. There can be additional modules in the current directory; you only need to specify the entry point. Given an executable stanza with (name <name>), Dune will know how to build <name>.exe. If requested, it will also know how to build <name>.bc, <name>.bc.js and <name>.bc.wasm.js (Dune 2.0 and up also needs specific configuration (see the modes optional field below)).

<name>.exe is a native code executable, <name>.bc is a bytecode executable which requires ocamlrun to run, <name>.bc.js is a JavaScript generated using js\_of\_ocaml, and <name>.bc.wasm.js is a Wasm loader script generated using wasm\_of\_ocaml (the Wasm modules are included in directory <name>.bc.wasm.assets).

Please note: in case native compilation is not available, <name>.exe will be a custom bytecode executable, in the sense of ocamlc -custom. This means it's a native executable that embeds the ocamlrun virtual machine as well as the bytecode, so you can always rely on <name>.exe being available. Moreover, it is usually preferable to use <name>.exe in custom rules or when calling the executable by hand because running a bytecode executable often requires loading

shared libraries that are locally built. This requires additional setup, such as setting specific environment variables, which Dune doesn't do at the moment.

Native compilation isn't available when there is no `ocamlc` binary at the same place as `ocamlc` was found.

Executables can also be linked as object or shared object files. See *linking modes* for more information.

Starting from Dune 3.0, it's possible to automatically generate empty interface files for executables. See *executables\_implicit\_empty\_intf*.

<optional-fields> are:

- (`public_name <public-name>`) specifies that the executable should be installed under this name. It's the same as adding the following stanza to your dune file:

```
(install
 (section bin)
 (files (<name>.exe as <public-name>)))
```

As a special case, (`public_name -`) is the same as if the field was absent.

- (`package <package>`) if there is a (`public_name ...`) field, this specifies the package the executables are part of it.
- (`libraries <library-dependencies>`) specifies the library dependencies. See *Library Dependencies* for more details.
- (`link_flags <flags>`) specifies additional flags to pass to the linker. This field supports (`:include ...`) forms.

Some dependencies require Dune to link executables with `-linkall` even if it is not present in `link_flags`. This is currently the case for `dune-site`, `dune-site.plugins`, and `findlib.dynload`.

- (`link_deps (<deps-conf list>)`) specifies the dependencies used only by the linker, i.e., when using a version script. See *Dependency Specification* for more details.
- (`modules <modules>`) specifies which modules in the current directory Dune should consider when building this executable. Modules not listed here will be ignored and cannot be used inside the executable described by the current stanza. It is interpreted in the same way as the (`modules ...`) field of *library*.
- (`root_module <module>`) specifies a `root_module` that collects all listed dependencies in `libraries`. See the documentation for `root_module` in the library stanza.
- (`modes (<modes>)`) sets the *linking modes*. The default is `(exe)`. Before Dune 2.0, it formerly was `(byte exe)`.
- (`preprocess <preprocess-spec>`) is the same as the (`preprocess ...`) field of *library*.
- (`lint <preprocess-spec>`) is the same as the (`lint ...`) field of *library*.
- (`preprocessor_deps (<deps-conf list>)`) is the same as the (`preprocessor_deps ...`) field of *library*.
- `js_of_ocaml`: See the section about *js\_of\_ocaml*
- `wasm_of_ocaml`: See the section about *wasm\_of\_ocaml*
- `flags`, `ocamlc_flags`, and `ocamlc_flags`: See *OCaml Flags*.
- (`modules_without_implementation <modules>`) is the same as the corresponding field of *library*.
- (`allow_overlapping_dependencies`) is the same as the corresponding field of *library*.
- (`optional`) is the same as the corresponding field of *library*.
- (`enabled_if <blang expression>`) is the same as the corresponding field of *library*.

- (`promote <options>`) allows promoting the linked executables to the source tree. The options are the same as for the *rule promote mode*. Adding (`promote (until-clean)`) to an `executable` stanza will cause Dune to copy the `.exe` files to the source tree and use `dune clean` to delete them.
- (`foreign_stubs <foreign-stubs-spec>`) specifies foreign source files, e.g., C or C++ stubs, to be linked into the executable. See *Foreign Stubs* for more details.
- (`foreign_archives <foreign-archives-list>`) specifies archives of foreign object files to be linked into the executable. See the section *Foreign Archives* for more details.
- (`forbidden_libraries <libraries>`) ensures that the given libraries are not linked in the resulting executable. If they end up being pulled in, either through a direct or transitive dependency, Dune fails with an error message explaining how the library was pulled in. This field has been available since Dune 2.0.
- (`embed_in_plugin_libraries <library-list>`) specifies a list of libraries to link statically when using the `plugin` linking mode. By default, no libraries are linked in. Note that you may need to also use the `-linkall` flag if some of the libraries listed here are not referenced from any of the plugin modules.
- (`ctypes <ctypes field>`) instructs Dune to use `ctypes stubgen` to process your type and function descriptions for binding system libraries, vendored libraries, or other foreign code. See *Stub Generation with Dune Ctypes* for a full reference. This field is available since the 3.0 version of the Dune language.
- (`empty_module_interface_if_absent`) causes the generation of empty interfaces for every module that does not have an interface file already. Useful when modules are used solely for their side-effects. This field is available since the 3.0 version of the Dune language.

## Linking Modes

The `modes` field allows selecting which linking modes will be used to link executables. Most modes are pairs (`<compilation-mode> <binary-kind>`), where `<compilation-mode>` describes whether the bytecode or native code backend of the OCaml compiler should be used and `<binary-kind>` describes what kind of file should be produced. The special mode `byte_complete` is described below.

`<compilation-mode>` must be `byte`, `native`, or `best`, where `best` is `native` with a fallback to bytecode when native compilation isn't available.

`<binary-kind>` is one of:

- `c` for producing OCaml bytecode embedded in a C file
- `exe` for normal executables
- `object` for producing static object files that can be manually linked into C applications
- `shared_object` for producing object files that can be dynamically loaded into an application. This mode can be used to write a plugin in OCaml for a non-OCaml application.
- `js` for producing JavaScript from bytecode executables, see *explicit\_js\_mode*.
- `wasm` for producing JavaScript from bytecode executables.
- `plugin` for producing a plugin (`.cmxs` if native or `.cma` if bytecode).

For instance the following `executables` stanza will produce bytecode executables and native shared objects:

```
(executables
 (names a b c)
 (modes (byte exe) (native shared_object)))
```

Additionally, you can use the following shorthands:

- `c` for `(byte c)`

- `exe` for `(best exe)`
- `object` for `(best object)`
- `shared_object` for `(best shared_object)`
- `byte` for `(byte exe)`
- `native` for `(native exe)`
- `js` for `(byte js)`
- `wasm` for `(byte wasm)`
- `plugin` for `(best plugin)`

For instance, the following modes fields are all equivalent:

```
(modes (exe object shared_object))
(modes ((best exe)
        (best object)
        (best shared_object)))
```

Use the special mode `byte_complete` for building a bytecode executable as a native self-contained executable, i.e., an executable that doesn't require the `ocamlrun` program to run and doesn't require the C stubs to be installed as shared object files. This mode has been available since Dune 3.6.

The extensions for the various linking modes are chosen as follows:

Linking mode	Extension
<code>byte</code>	<code>.bc</code>
<code>native</code> or <code>best</code>	<code>.exe</code>
<code>byte_complete</code>	<code>.bc.exe</code>
<code>(byte object)</code>	<code>.bc%{ext_obj}</code>
<code>(native object)</code> or <code>(best object)</code>	<code>.exe%{ext_obj}</code>
<code>(byte shared_object)</code>	<code>.bc%{ext_dll}</code>
<code>(native shared_object)</code> or <code>(best shared_object)</code>	<code>%{ext_dll}</code>
<code>c</code>	<code>.bc.c</code>
<code>js</code>	<code>.bc.js</code>
<code>wasm</code>	<code>.bc.wasm.js</code>
<code>(best plugin)</code>	<code>%{ext_plugin}</code>
<code>(byte plugin)</code>	<code>.cma</code>
<code>(native plugin)</code>	<code>.cmxs</code>

`%{ext_obj}` and `%{ext_dll}` are the extensions for object and shared object files. Their value depends on the OS. For instance, on Unix `%{ext_obj}` is usually `.o` and `%{ext_dll}` is usually `.so`, while on Windows `%{ext_obj}` is `.obj` and `%{ext_dll}` is `.dll`.

When `byte` is specified but neither `native` nor `exe` is specified, Dune also builds a self-contained bytecode executable with the `.exe` extension for compatibility. This is similar to `byte_complete`, but uses the `.exe` extension rather than `.bc.exe`. `.bc` files require additional files at runtime that aren't currently tracked by Dune, so Dune doesn't run `.bc` files during the build. Run the `.bc.exe` or `.exe` targets instead, as these are self-contained.

Lastly, note that `.bc` executables cannot contain C stubs. If your executable contains C stubs you may want to use `(modes exe)`.

## js\_of\_ocaml

In library and executable stanzas, you can specify `js_of_ocaml` options using `(js_of_ocaml (<js_of_ocaml-options>))`.

`<js_of_ocaml-options>` are all optional:

- `(flags <flags>)` to specify flags passed to `js_of_ocaml compile`. This field supports `(:include ...)` forms
- `(build_runtime_flags <flags>)` to specify flags passed to `js_of_ocaml build-runtime`. This field supports `(:include ...)` forms
- `(link_flags <flags>)` to specify flags passed to `js_of_ocaml link`. This field supports `(:include ...)` forms
- `(javascript_files (<files-list>))` to specify `js_of_ocaml` JavaScript runtime files.
- `(compilation_mode <mode>)` where `<mode>` is either `whole_program` or `separate`. This is only available inside executable stanzas.
- `(sourcemap <config>)` where `<config>` is one of `no`, `file` or `inline`. This is only available inside executable stanzas.
- `(enabled_if <blang expression>)` to specify whether the `js` mode is enabled. It is enabled by default. This is only available inside executable stanzas.

`<flags>` is specified in the *Ordered Set Language*. `<blang expression>` is specified using the *Boolean Language*,

The default values for `flags`, `compilation_mode` and `sourcemap` depend on the selected build profile. The build profile `dev` (the default) will enable inline sourcemap, separate compilation and pretty JavaScript output.

See *JavaScript Compilation With Js\_of\_ocaml* for more information.

## wasm\_of\_ocaml

In library and executable stanzas, you can specify `wasm_of_ocaml` options using `(wasm_of_ocaml (<wasm_of_ocaml-options>))`.

`<wasm_of_ocaml-options>` are all optional. They are the same as the `<js_of_ocaml-options>` above plus:

- `(wasm_files (<files-list>))` to specify `wasm_of_ocaml` Wasm runtime files.

For the `(sourcemap <config>)` option, `<config>` must be one of `no` or `inline`. Source maps are put within the `.bc.wasm.assets` directory.

The default values for `flags`, `compilation_mode` and `sourcemap` depend on the selected build profile. The build profile `dev` (the default) will enable sourcemaps, separate compilation and pretty Wasm output.

See *Wasm Compilation With Wasm\_of\_ocaml* for more information.

### 4.1.2 executables

There is a very subtle difference in the naming of these stanzas. One is `executables`, plural, and the other is `executable`, singular. The `executables` stanza is very similar as the `executable` stanza but can be used to describe several executables sharing the same configuration, so the plural `executables` stanza is used to describe more than one executable.

It shares the same fields as the `executable` stanza, except that instead of `(name ...)` and `(public_name ...)` you must use the plural versions as well:

- `(names <names>)` where `<names>` is a list of entry point names. Compare with `executable`, where you only need to specify the modules containing the entry point of each executable.

- `(public_names <names>)` describes under what name to install each executable. The list of names must be of the same length as the list in the `(names ...)` field. Moreover, you can use `-` for executables that shouldn't be installed.

However, using `executables` the executables defined in the stanza are allowed to share modules.

Given modules `Foo`, `Bar` and `Baz` the usage of `executables` can simplify the code:

```
(executables
 (names foo bar))
```

Instead of the more complex

```
(library
 (name baz)
 (modules baz))

(executable
 (name foo)
 (modules foo)
 (libraries baz))

(executable
 (name bar)
 (modules bar)
 (libraries baz))
```

### 4.1.3 library

The `library` stanza must be used to describe OCaml libraries. The format of `library` stanzas is as follows:

```
(library
 (name <library-name>)
 <optional-fields>)
```

`<library-name>` is the real name of the library. It determines the names of the archive files generated for the library as well as the module name under which the library will be available, unless `(wrapped false)` is used (see below). It must be a valid OCaml module name, but it doesn't need to start with an uppercase letter.

For instance, the modules of a library named `foo` will be available as `Foo.XXX`, outside of `foo` itself; however, it is allowed to write an explicit `Foo` module, which will be the library interface. You are free to expose only the modules you want.

Please note: by default, libraries and other things that consume OCaml/Reason modules only consume modules from the directory where the stanza appear. In order to declare a multi-directory library, you need to use the `include_subdirs` stanza.

`<optional-fields>` are:

**(public\_name <name>)**

The name under which the library can be referred as a dependency when it's not part of the current workspace, i.e., when it's installed. Without a `(public_name ...)` field, the library won't be installed by Dune. The public name must start with the package name it's part of and optionally followed by a dot, then anything else you want. The package name must also be one of the packages that Dune knows about, as determined by the logic described in *Packages*.

**(package <package>)**

Installs a private library under the specified package. Such a library is now usable by public libraries defined in the same project. The Findlib name for this library will be `<package>.__private__.<name>`; however, the library's interface will be hidden from consumers outside the project.

**(synopsis <string>)**

A one-line description of the library. This is used by tools that list installed libraries.

**(modules <modules>)**

Specifies what modules are part of the library. By default, Dune will use all the `.ml/.re` files in the same directory as the dune file. This includes ones present in the file system as well as ones generated by user rules. You can restrict this list by using a `(modules <modules>)` field.

`<modules>` uses the *Ordered Set Language*, where elements are module names and don't need to start with an uppercase letter. For instance, to exclude module `Foo`, use `(modules (:standard \ foo))`.

Starting in Dune 3.13, one can also use special forms `(:include <file>)` and variables such as `%{read-lines:<file>}` in this field to customize the list of modules using Dune rules. The dependencies introduced in this way *must live in a different directory that the stanza making use of them*.

**(libraries <library-dependencies>)**

Specifies the library's dependencies.

See *Library Dependencies* for more details.

**(wrapped <boolean>)**

Specifies whether the library modules should be available only through the top-level library module, or if they should all be exposed at the top level.

The default is `true`, and it's highly recommended to keep it this way. Because OCaml top-level modules must all be unique when linking an executables, polluting the top-level namespace will make your library unusable with other libraries if there is a module name clash.

This option is only intended for libraries that manually prefix all their modules by the library name and to ease porting of existing projects to Dune.

**(wrapped (transition <message>))**

This is the same as `(wrapped true)`, except it will also generate unwrapped (not prefixed by the library name) modules to preserve compatibility.

This is useful for libraries that would like to transition from `(wrapped false)` to `(wrapped true)` without breaking compatibility for users. The deprecation notices for the unwrapped modules will include `<message>`.

**(preprocess <preprocess-spec>)**

Specifies how to preprocess files when needed.

The default is `no_preprocessing`, and other options are described in *Preprocessing Specification*.

**(lint <preprocess-spec>)**

Specifies how to lint source files when building the `@lint` alias.

The default is `no_preprocessing`. The syntax is described in *Linting*.

**(preprocessor\_deps (<deps-conf list>))**

Specifies extra preprocessor dependencies preprocessor, i.e., if the preprocessor reads a generated file.

The specification of dependencies is described in *Dependency Specification*.

### **(optional)**

If present, it indicates that the library should only be built and installed if all the dependencies are available, either in the workspace or in the installed world.

Use this to provide extra features without adding hard dependencies to your project.

### **(foreign\_stubs <foreign-stubs-spec>)**

Specifies foreign source files, e.g., C or C++ stubs, to be compiled and packaged together with the library.

See the section *Foreign Stubs* for more details.

This field replaces the now-deleted fields `c_names`, `c_flags`, `cxx_names`, and `cxx_flags`.

### **(foreign\_archives <foreign-archives-list>)**

Specifies archives of foreign object files to be packaged with the library.

See the section *Foreign Archives* for more details. This field replaces the now-deleted field `self_build_stubs_archive`.

### **(install\_c\_headers (<names>))**

If your library has public C header files that must be installed, you must list them in this field, without the `.h` extension.

You should favor the `public_headers` field starting from 3.8.

### **(public\_headers (<files>))**

If your library has public C header files that must be installed, you must list them in this field. This field accepts globs in the form of `(glob_files_rec <glob>)` and `(glob_files <glob>)` fields to specify multiple files.

The advantage of this field over `install_c_headers` is that it preserves the directory structures of the headers relative to the library stanza. Additionally, it allows to specify the extensions of the header files, which allows alternative extensions such as `.hh` or `.hpp`.

### **(modes <modes>)**

List modes which should be built by default.

The most common use for this feature is to disable native compilation when writing libraries for the OCaml toplevel.

The following modes are available: `byte`, `native` and `best`. `best` is `native` or `byte` when native compilation isn't available.

### **(no\_dynlink)**

Disables (native) dynamic linking of the library. This means that the `.cmxs` archive of the library will neither be built nor installed.

This is for advanced use only. By default, you shouldn't set this option.

### **(kind <kind>)**

Sets the type of library.

The default is `normal`, but other available choices are `ppx_rewriter` and `ppx_deriver`.

They must be set when the library is intended to be used as a PPX rewriter or a `[@@deriving ...]` plugin. The reason `ppx_rewriter` and `ppx_deriver` are split is historical, and hopefully we won't need two options soon.

Both PPX kinds support an optional field: `(cookies <cookies>)`, where `<cookies>` is a list of pairs (`<name>` `<value>`) with `<name>` being the cookie name and `<value>` a string that supports *Variables* evaluated by each preprocessor invocation (note: libraries that share cookies with the same name should agree on their expanded value).

**(ppx\_runtime\_libraries (<library-names>))**

This field is for when the library is a ppx rewriter or a [@@deriving ...] plugin, and has runtime dependencies.

You need to specify these runtime dependencies here.

**(virtual\_deps (<opam-packages>))**

Sometimes opam packages enable a specific feature only if another package is installed. For instance, the case of ctypes will only install ctypes.foreign if the dummy ctypes-foreign package is installed.

You can specify such virtual dependencies here, but you don't need to do so unless you use Dune to synthesize the depends and depopts sections of your opam file.

**(implements <name>)**

name defines the name of the virtual library or the parameter library you are implementing.

See *Virtual Libraries* or *library\_parameter*.

**(parameters <library-parameter-names>)**

List the library parameters used by the library and its dependencies.

This feature is experimental and requires the compiler you are using to support parameterised libraries. See *library\_parameter*.

**(js\_of\_ocaml ...)**

Sets options for JavaScript compilation, see *js\_of\_ocaml*.

**(wasm\_of\_ocaml ...)**

Sets options for JavaScript compilation, see *wasm\_of\_ocaml*.

**(flags ...)**

See *OCaml Flags*.

**(ocamlc\_flags ...)**

See *OCaml Flags*.

**(ocamlopt\_flags ...)**

See *OCaml Flags*.

**(library\_flags (<flags>))**

A list of flags passed to ocamlc and ocamlopt when building the library archive files.

You can use this to specify -linkall, for instance.

<flags> is a list of strings supporting *Variables*.

**(c\_library\_flags <flags>)**

Specifies the flags passed to the C compiler when constructing the library archive file for the C stubs.

<flags> uses the *Ordered Set Language* and supports (:include ...) forms.

When you write bindings for a C library named bar, you should typically write -lbar here, or whatever flags are necessary to link against this library.

**(modules\_without\_implementation <modules>)**

Specifies a list of modules that have only a .mli or .rei but no .ml or .re file.

Such modules are usually referred as *mli only modules*. They are not officially supported by the OCaml compiler; however, they are commonly used. Such modules must only define types.

Since it isn't reasonably possible for Dune to check this is the case, Dune requires the user to explicitly list such modules to avoid surprises.

Note that the `modules_without_implementation` field isn't merged in `modules`, which represents the total set of modules in a library. If a directory has more than one stanza, and thus a `modules` field must be specified, `<modules>` still needs to be added in `modules`.

### **(private\_modules <modules>)**

Specifies a list of modules that will be marked as private.

Private modules are inaccessible from outside the libraries they are defined in.

Note that the `private_modules` field is not merged in `modules`, which represents the total set of modules in a library. If a directory has more than one stanza and thus a `modules` field must be specified, `<modules>` still need to be added in `modules`.

### **(allow\_overlapping\_dependencies)**

Allows external dependencies to overlap with libraries that are present in the workspace.

### **(enabled\_if <blang expression>)**

Conditionally disables a library.

A disabled library cannot be built and will not be installed.

The condition is specified using the *Boolean Language*, and the field allows for the `%{os_type}` variable, which is expanded to the type of OS being targeted by the current build. Its value is the same as the value of the `os_type` parameter in the output of `ocamlc -config`.

### **(inline\_tests)**

Enables inline tests for this library.

They can be configured through options using `(inline_tests <options>)`.

See *Inline Tests* for a reference of corresponding options.

### **(root\_module <module>)**

This field instructs Dune to generate a module that will contain module aliases for every library specified in dependencies.

This is useful whenever a library is shadowed by a local module. The library may then still be accessible via this root module.

### **(ctypes <ctypes field>)**

Instructs Dune to use `ctypes stubgen` to process your type and function descriptions for binding system libraries, vendored libraries, or other foreign code.

See *Stub Generation with Dune Ctypes* for a full reference.

This field is available since the 3.0 version of the Dune language.

### **(empty\_module\_interface\_if\_absent)**

Causes the generation of empty interfaces for every module that does not have an interface file already.

Useful when modules are used solely for their side-effects. This field is available since the 3.0 version of the Dune language.

Note that when binding C libraries, Dune doesn't provide special support for tools such as `pkg-config`; however, it integrates easily with *Configurator* by using `(c_flags (:include ...))` and `(c_library_flags (:include ...))`.

### 4.1.4 `foreign_library`

The `foreign_library` stanza describes archives of separately compiled foreign object files that can be packaged with an OCaml library or linked into an OCaml executable. See *Foreign Archives* for further details and examples.

### 4.1.5 `deprecated_library_name`

The `deprecated_library_name` stanza enables redirecting an old deprecated name after a library has been renamed. Its syntax is as follows:

```
(deprecated_library_name
 (old_public_name <name>)
 (new_public_name <name>))
```

When a developer uses the old public name in a list of library dependencies, it will be transparently replaced by the new name. Note that it's not necessary for the new name to exist at definition time, as it is only resolved at the point where the old name is used.

The `old_public_name` can also be one of the names declared in the `deprecated_package_names` field of the package declaration in the `dune-project` file. In this case, the "old" library is understood to be a library whose name is not prefixed by the package name. Such a library cannot be defined in Dune, but other build systems allow it. This feature is meant to help migration from those systems.

### 4.1.6 `generate_sites_module`

Added in version 2.8.

Dune proposes some facilities for dealing with *sites* in a program. The `generate_sites_module` stanza will generate code for looking up the correct locations of the sites' directories and for loading plugins. It works after installation with or without the relocation mode, inside Dune rules, and when using Dune executables. For promotion, it works only if the generated modules are solely in the executable (or library statically linked) promoted; generated modules in plugins won't work.

```
(generate_sites_module
 (module <name>)
 <facilities>)
```

The module's code is generated in the directory with the given name. The code is populated according to the requested facilities.

The available `<facilities>` are:

- `sourceroot` adds a value `val sourceroot: string option` in the generated module, which contains the value of `workspace_root`, if the code has been built locally. It could be used to keep the tool's configuration file locally when executed with `dune exec` or after promotion. The value is `None` once it has been installed.

The generated code also recognizes the `DUNE_SOURCEROOT` environment variable as a fallback when no source root was embedded in the module. Dune sets this variable to the workspace root when it sets up the runtime environment for sites. In Dune rules, prefer the `workspace_root` or `project_root` variables when you need the source tree location.

- `relocatable` adds a value `val relocatable: bool` in the generated module, which indicates if the binary has been installed in the relocatable mode.
- `(sites <package>)` adds a value `val <site>: string list` for each `<site>` of `<package>` in the submodule `Sites` of the generated module. The identifier `<site>` isn't capitalized.
- `(plugins (<package> <site>) ...)` adds a submodule `<site>` with the following signature `S` in the submodule `Plugins` of the generated module. The identifier `<site>` is capitalized.

```

module type S = sig
  val paths: string list
  (** return the locations of the directory containing the plugins *)

  val list: unit -> string list
  (** return the list of available plugins *)

  val load_all: unit -> unit
  (** load all the plugins and their dependencies *)

  val load: string -> unit
  (** load the specified plugin and its dependencies *)
end

```

The generated module is a dependency on the library `dune-site`, and if the facilities (`plugins ...`) are used, it is a dependency on the library `dune-site.plugins`. Those dependencies are not automatically added to the library or executable which use the module (cf. *Plugins and Dynamic Loading of Packages*).

### 4.1.7 test

The test stanza is the singular form of `tests`. The only difference is that it's of the form:

```

(test
  (name foo)
  <optional fields>)

```

The name field is singular, and the same optional fields are supported.

### 4.1.8 tests

The `tests` stanza allows one to easily define multiple tests. For example, we can define two tests at once with:

```

(tests
  (names mytest expect_test)
  <optional fields>)

```

This defines an executable named `mytest.exe`. These tests can be run by building the aliases `runtest-mytest` and `runtest-expect_test` respectively. They will also be added to the `runtest` alias.

If the directory also contains an `expect_test.expected` file, then `expect_test` will be used to define an expect test. That is, the test will be executed and its output will be compared to `expect_test.expected`.

The optional fields supported are a subset of the alias and executables fields. In particular, all fields except for `public_names` are supported from the *executables stanza*. Alias fields apart from `name` are allowed.

The `(enabled_if)` field has special semantics: when present, it only applies to running the tests. The test executable is always built by default. If you need to restrict building the test executable, use `(build_if)` instead.

By default, the test binaries are run without options. The `action` field can override the test binary invocation, i.e., if you're using `Alcotest` and wish to see all the test failures on the standard output. When running Dune `runtest` you can use the following stanza:

```

(tests
  (names mytest)
  (libraries alcotest mylib)
  (action (run %{test} -e)))

```

Starting from Dune 2.9, it's possible to automatically generate empty interface files for test executables. See *executables\_implicit\_empty\_intf*.

### 4.1.9 Cram

#### (cram ...)

Configure Cram tests in the current directory (and subdirectories).

A single test may be configured by more than one `cram` stanza. In such cases, the values from all applicable `cram` stanzas are merged together to get the final values for all the fields.

#### ➔ See also

*Cram Tests*

#### (deps <dep-spec>)

Specify the dependencies of the test.

When testing binaries, it's important to specify a dependency on the binary for two reasons:

- Dune must know to re-run the test when a dependency changes
- The dependencies must be specified to guarantee that they're visible to the test when running it.

The following introduces a dependency on `foo.exe` on all Cram tests in this directory:

```
(cram
 (deps ../foo.exe))
```

#### ➔ See also

*Dependency Specification.*

#### (applies\_to <predicate-lang>)

Specify the scope of this `cram` stanza. By default it applies to all the Cram tests in the current directory. The special `:whole_subtree` value will apply the options to all tests in all subdirectories (recursively). This is useful to apply common options to an entire test suite.

The following will apply the stanza to all tests in this directory, except for `foo.t` and `bar.t`:

```
(cram
 (applies_to * \ foo bar)
 (deps ../foo.exe))
```

#### ➔ See also

*Predicate Language*

#### (enabled\_if <blang>)

Control whether the tests are included in the `runtest` alias and other aliases specified by the `alias` field. When `enabled_if` evaluates to `false`, the test is excluded from these aliases but can still be run explicitly via its own alias (e.g., `dune build @foo` for `foo.t`).

**➔ See also**

*Boolean Language, Variables*

**(alias <name>)**

Alias that can be used to run the test. In addition to the user alias, every test `foo.t` is attached to the `@runtest` alias and gets its own `@foo` alias to make it convenient to run individually.

**(locks <lock-names>)**

Specify that the tests must be run while holding the following locks.

**➔ See also**

*Locks*

**(package <name>)**

Attach the tests selected by this stanza to the specified package.

**(runtest\_alias <true|false>)**

Added in version 3.12.

When set to `false`, do not add the tests to the `runtest` alias. The default is to add every Cram test to `runtest`, but this is not always desired.

**(timeout <float>)**

Added in version 3.20.

Specify a time limit (in seconds) for each individual Cram test.

If a test takes longer than the specified timeout, Dune will terminate it and report a timeout error. This can be useful to catch tests that hang or take unexpectedly long.

The timeout is a floating-point number (e.g., `1.5` for 1.5 seconds). Zero or negative values cause immediate failure when running the cram test.

If multiple `cram` stanzas apply to the same test, the **lowest** of all specified timeouts is used.

This field is typically used to guard against unresponsive or non-terminating test cases.

Example:

```
(cram
 (timeout 2.5))
```

This limits each selected test to at most 2.5 seconds of execution time.

**(conflict\_markers <ignore|error>)**

Added in version 3.21.

Determines how conflict markers inserted by version control systems are inserted. The default behavior is to `ignore` them. Setting `error` will make the test runner reject such conflicts and refuse to run the test.

**(setup\_scripts <files-list>)**

Added in version 3.21.

Determines the list of shell files that will be sourced before the execution of every `cram` test this stanza is applied to. A single script will not be sourced more than once, no matter how many times it is listed in this field. The order of evaluation for these scripts is deterministic but is left undefined, so it is not recommended that these scripts have side effects.

`(shell <sh|bash>)`

Added in version 3.22.

Determines the shell executable to use to execute the cram script. By default, cram scripts will execute with `sh`.

#### 4.1.10 toplevel

The `toplevel` stanza allows one to define custom toplevels. Custom toplevels automatically load a set of specified libraries and are runnable like normal executables. Example:

```
(toplevel
 (name tt)
 (libraries str))
```

This will create a toplevel with the `str` library loaded. We may build and run this toplevel with:

```
$ dune exec ./tt.exe
```

`(preprocess (pps ...))` is the same as the `(preprocess (pps ...))` field of *library*. Currently, `action` and `future_syntax` are not supported in the toplevel.

#### 4.1.11 documentation

Additional manual pages may be attached to packages using the `documentation` stanza. These `.mld` files must contain text in the same syntax as OCaml doc comments.

```
(documentation (<optional-fields>))
```

Where `<optional-fields>` are:

- `(package <name>)` defines the package this documentation should be attached to. If this is absent, Dune will try to infer it based on the location of the stanza.
- `(mld_files <arg>)`: the `<arg>` field follows the *Ordered Set Language*. This is a set of extensionless MLD file basenames attached to the package, where `:standard` refers to all the `.mld` files in the stanza's directory.
- `(files <arg>)`: the `files` field accepts the same arguments as the one from the *install stanza*. It allows to install `mld` files as well as asset files, and specify where they are in the hierarchy of documentation (with the `as` and `with_prefix` keyword). Note that dune supports installing those files, but not yet building the documentation with a non-flat hierarchy, or with non-`mld` files.

For more information, see *Generating Documentation*.

#### 4.1.12 install

Dune supports installing packages on the system, i.e., copying freshly built artifacts from the workspace to the system. The `install` stanza takes three pieces of information:

- The list of files or directories to install
- The package to attach these files. This field is optional if your project contains a single package.
- The section in which the files will be installed

For instance:

```
(install
 (files hello.txt)
 (section share)
 (package mypackage))
```

Indicate that the file `hello.txt` in the current directory is to be installed in `<prefix>/share/mypackage`.

The following sections are available:

Section	Target	Remarks
<code>lib</code>	<code>&lt;prefix&gt;/lib/&lt;pkgname&gt;/</code>	
<code>lib_root</code>	<code>&lt;prefix&gt;/lib/</code>	
<code>libexec</code>	<code>&lt;prefix&gt;/lib/&lt;pkgname&gt;/</code>	executable bit is set
<code>libexec_root</code>	<code>&lt;prefix&gt;/lib/</code>	executable bit is set
<code>bin</code>	<code>&lt;prefix&gt;/bin/</code>	executable bit is set
<code>sbin</code>	<code>&lt;prefix&gt;/sbin/</code>	executable bit is set
<code>toplevel</code>	<code>&lt;prefix&gt;/lib/toplevel/</code>	
<code>share</code>	<code>&lt;prefix&gt;/share/&lt;pkgname&gt;/</code>	
<code>share_root</code>	<code>&lt;prefix&gt;/share/</code>	
<code>etc</code>	<code>&lt;prefix&gt;/etc/&lt;pkgname&gt;/</code>	
<code>stublibs</code>	<code>&lt;prefix&gt;/lib/stublibs/</code>	executable bit is set
<code>doc</code>	<code>&lt;prefix&gt;/doc/&lt;pkgname&gt;/</code>	
<code>man</code>	<code>&lt;prefix&gt;/man/manX/</code>	(see below)
<code>misc</code>	absolute destination	(see below)
<code>(site (&lt;package&gt; &lt;site&gt;))</code>	<code>&lt;site&gt;</code> directory of <code>&lt;package&gt;</code>	(see below)

Additional remarks:

- For `man`, the exact destination is inferred from the file extension. For example, `foo.1` is installed as `<prefix>/man/man1/foo.1`.
- `misc` only works when using `opam`. In that case, the user will be prompted before installation. This mechanism is deprecated.
- In the case of `(site)`, if the prefix isn't the same as the one used when installing `<package>`, `<package>` won't find the files.

Normally, Dune uses the file's basename to determine the file's name once installed; however, you can change that by using the form `(<filename> as <destination>)` in the `files` field. For instance, to install a file `mylib.el` as `<prefix>/emacs/site-lisp/mylib.el`, you must write the following:

```
(install
 (section share_root)
 (files (mylib.el as emacs/site-lisp/mylib.el)))
```

The mode of installed files is fully determined by the section they are installed in. If the section above is documented as with the executable bit set, they are installed with mode `0o755 (rwxr-xr-x)`; otherwise they are installed with mode `0o644 (rw-r--r--)`.

Note that all files in the install stanza must be specified by relative paths only. It is an error to specify files by absolute paths.

Also note that as of `dune-lang 3.11` (i.e., `(lang dune 3.11)` in `dune-project`) it is deprecated to use the `as` keyword to specify a destination beginning with `..`. Dune intends for files associated with a package to only be installed under specific directories in the file system implied by the installation section (e.g., `share`, `bin`, `doc`, etc.) and the package name. Starting destination paths with `..` allows packages to install files to arbitrary locations on the file system. In

3.11, this behaviour is still supported (as some projects may depend on it) but will generate a warning and will be removed in a future version of Dune.

### Including Files in the Install Stanza

You can include external files from the `files` and `dirs` fields of the install stanza:

```
(install
 (files (include foo.sexp))
 (section share))
```

Here the file `foo.sexp` must contain a single S-expression list, whose elements will be included in the list of files or directories to install. That is, elements may be of the form:

- `<filename>`
- `(<filename> as <destination>)`
- `(include <filename>)`

Included files may be generated by rules. Here is an example of a rule which generates a file by listing all the files in a subdirectory `resources`:

```
(rule
 (deps (source_tree resources))
 (action
 (with-stdout-to foo.sexp
 (system "echo '(' resources/* ')')"))))
```

### Globs in the Install Stanza

You can use globs to specify files to install by using the terms `(glob_files <glob>)` and `(glob_files_rec <glob>)` inside the `files` field of the install stanza (but not inside the `dirs` field). See the [glob](#) for details of the glob syntax. The `(glob_files <glob>)` term will expand its argument within a single directory, whereas the `(glob_files_rec <glob>)` term will recursively expand its argument within all subdirectories.

For example:

```
(install
 (files
 (glob_files style/*.css)
 (glob_files_rec content/*.html))
 (section share))
```

This example will install:

- All files matching `*.css` in the `style` directory.
- All files matching `*.html` in the `content` directory, or any of its descendant subdirectories.

Note that the paths to files are preserved after installation. Suppose the source directory contained the files `style/foo.css` and `content/bar/baz.html`. The example above will place these files in `share/<package>/style/foo.css` and `share/<package>/content/bar/baz.html` respectively where `<package>` is the name of the package (ie. `dune-project` would contain `(package (name <package>))`).

The `with_prefix` keyword can be used to change the destination path of files matched by a glob, similar to the `as` keyword in the `(files ...)` field. `with_prefix` changes the prefix of a path before the component matched by the `*` to some new value. For example:

```
(install
 (files
  (glob_files (style/*.css with_prefix web/stylesheets))
  (glob_files_rec (content/*.html with_prefix web/documents)))
 (section share))
```

Continuing the example above, this would result in the source file at `style/foo.css` being installed to `share/<package>/web/stylesheets/foo.css` and `content/bar/baz.html` being installed to `share/<package>/web/documents/bar/baz.html`. Note in the latter case `with_prefix` only replaced the `content` component of the path and not the `bar` component since it replaces the prefix of the glob - not the prefix of paths matching the glob.

### Installing Globs from Parent Directories

The default treatment of paths in globs creates a complication where referring to globs in a parent directory such as `(glob_files ../*.txt)` would attempt to install the matched files outside the designated install directory. For example writing:

```
(install
 (files (glob_files ../*.txt))
 (section share))
```

... would cause Dune to attempt to install the matching files to `share/<package>/../`, ie. `share` where `<package>` is the name of the package (i.e., `dune-project` would contain `(package (name <package>))`). This is probably not what the user intends, and installing files to relative paths beginning with `..` is deprecated from version 3.11 of Dune and will become an error in a future version.

The solution is to use `with_prefix` to replace the `..` with some other path. For example:

```
(install
 (files (glob_files ../*.txt with_prefix .))
 (section share))
```

... would install the matched files to `share/<package>/` instead.

### Handling of the .exe Extension on Windows

Under Microsoft Windows, executables must be suffixed with `.exe`. Dune tries to ensure that executables are always installed with this extension on Windows.

More precisely, when installing a file via an `(install ...)` stanza, Dune implicitly adds the `.exe` extension to the destination, if the source file has extension `.exe` or `.bc` and if it's not already present

### Installing Source Directories

To install entire source directories, the `source_tree` field can be used:

```
(install
 (section doc)
 (source_trees manual))
```

This example results in the contents of the `manual` directory being installed under `<prefix>/doc/<package>/manual/`.

As with `(files ...)` the destination can be changed with the `as` keyword. For example if you want to install all the files in the `manual` directory directly into `<prefix>/doc/<package>/` you can write:

```
(install
 (section doc)
 (source_trees (manual as .)))
```

It's also possible to specify multiple directories:

```
(install
 (section doc)
 (source_trees manual examples))
```

This would result in the local directories `manual` and `examples` being installed to `<prefix>/doc/<package>/manual/` and `<prefix>/doc/<package>/examples/` respectively.

Unlike with `(files ...)` it is an error to begin the destination (the right-hand side of `as`) with `...` (This is because support for installing source directories was added to Dune after destinations beginning with `..` were deprecated.)

### 4.1.13 plugin

Added in version 2.8.

Plugins are a way to load OCaml libraries at runtime. The `plugin` stanza allows you to declare the plugin's name, which *sites* should be present and which libraries it will load.

```
(plugin
 (name <name>)
 (libraries <libraries>)
 (site (<package> <site name>))
 (<optional-fields>))
```

`<optional-fields>` are:

- `(package <package>)` if there is more than one package defined in the current scope, this specifies which package the plugin will install. A plugin can be installed by one package in the site of another package.
- `(optional)` will not declare the plugin if the libraries are not available.

The loading of the plugin is done using the facilities generated by `generate_sites_module`.

### 4.1.14 rule

The `rule` stanza is used to create custom user rules. It tells Dune how to generate files from dependencies, or how to run an action when an alias is requested.

The syntax is as follows:

```
(rule
 (action <action>)
 <optional-fields>)
```

`<action>` is what you run to produce the targets from the dependencies, or to build the attached aliases. See *Actions* for more details.

A rule must either have at least one target or be attached to at least one alias. Targets may be listed explicitly with `target` or `targets`, or inferred from the action. Alias-only rules are useful for commands that don't produce files, such as benchmarks.

`<optional-fields>` are:

- `(target <filename>)` or `(targets <filenames>)` is a list of filenames (if defined with `targets`) or exactly one filename (if defined with `target`). Dune needs to statically know file targets of each rule. `(targets)` can be omitted if it can be inferred from the action. See *inferred rules*. It can also be omitted when the rule is only attached to an alias.
- `(deps <deps-conf list>)` specifies the dependencies of the rule. See *Dependency Specification* for more details.
- `(mode <mode>)` specifies how to handle the targets. See *modes* for details.
- `(fallback)` is deprecated and is the same as `(mode fallback)`.
- `(locks (<lock-names>))` specifies that the action must be run while holding the following locks. See *Locks* for more details.
- `(alias <alias-name>)` specifies this rule's alias. If the rule has file targets, building this alias builds those targets. If the rule has no file targets, building this alias runs the rule's action.
- `(aliases <alias-name list>)` specifies many aliases for this rule.
- `(package <package>)` specifies this rule's package. This rule will be unavailable when installing other packages in release mode.
- `(enabled_if <blang expression>)` specifies the Boolean condition that must be true for the rule to be considered. The condition is specified using the *Boolean Language*, and the field allows for *Variables* to appear in the expressions.
- `(corrections <setting>)` controls whether the rule's action may produce correction files that Dune reports as promotions. See *Corrections* for details. This field has been available since Dune 3.23.

Please note: contrary to makefiles or other build systems, user rules currently don't support patterns, such as a rule to produce `% .y` from `% .x` for any given `%`. This might be supported in the future.

After a rule's action exits, Dune records the produced targets. It must be able to read every target, including files inside directory targets. Dune also removes write permissions from generated files in the build directory before recording them. Actions may write their targets while they run, but generated targets should be treated as immutable once the rule finishes.

### Modes

By default, a rule's target must not exist in the source tree because Dune will error out when this is the case; however, it's possible to change this behavior using the `mode` field. The following modes are available:

- `standard` - the standard mode.
- `fallback` - in this mode, when the targets are already present in the source tree, Dune will ignore the rule. It's an error if only a subset of the targets are present in the tree. Fallback rules are commonly used to generate default configuration files that may be generated by a configure script.
- `promote` or `(promote <options>)` - in this mode, the files in the source tree will be ignored. Once the rule has been executed, the targets will be copied back to the source tree. The following options are available:
  - `(until-clean)` means that `dune clean` will remove the promoted files from the source tree.
  - `(into <dir>)` means that the files are promoted in `<dir>` instead of the current directory. This feature has been available since Dune 1.8.
  - `(only <predicate>)` means that only a subset of the targets should be promoted. The argument is similar to the argument of `subdir`, specified using the *Predicate Language*. This feature has been available since Dune 1.10.

There are two use cases for `promote` rules. The first one is when the generated code is easier to review than the generator, so it's easier to commit the generated code and review it. The second is to cut down dependencies during

releases. By passing `--ignore-promoted-rules` to Dune, rules with `(mode promote)` will be ignored, and the source files will be used instead. The `-p/--for-release-of-packages` flag implies `--ignore-promote-rules`. However, rules that promote only a subset of their targets via `(only ...)` are never ignored.

## Corrections

Rules can opt in to correction discovery with the `corrections` field. This is useful for custom tools that rewrite, format, lint, or otherwise suggest updates to source files without using an explicit `(diff? ...)` action.

The field accepts the following settings:

- `ignore` is the default. Dune does not look for correction files produced by the rule.
- `produce` makes Dune run the action in a sandbox and, when the action finishes, scan the sandbox for files ending in `.corrected`. A file named `foo.corrected` is treated as the corrected version of `foo`. If the files differ, Dune prints a diff, fails the rule, and records a promotion from `foo.corrected` to `foo`. The correction can then be applied with `dune promote` or `dune runtest --auto-promote`. If the corrected file is identical to the source file, no promotion is recorded and the rule succeeds.

For example:

```
(rule
  (alias runtest)
  (deps expected-output)
  (corrections produce)
  (action
    (run ./check-and-rewrite.exe expected-output expected-output.corrected)))
```

The same naming convention applies in subdirectories: `subdir/foo.corrected` is a correction for `subdir/foo`. Correction files that are themselves dependencies of the rule are ignored, so copied dependencies are not promoted by accident.

`(corrections produce)` cannot be combined with `patch_back_source_tree` sandboxing.

## Inferred Rules

When using the action DSL (see *Actions*), the dependencies and targets are usually obvious.

For instance:

```
(rule
  (target b)
  (deps a)
  (action (copy %{deps} %{target})))
```

In this example, the dependencies and targets are obvious by inspecting the action. When this is the case, you can use the following shorter syntax and have Dune infer dependencies and targets for you:

```
(rule <action>)
```

For instance:

```
(rule (copy a b))
```

Note that in Dune, targets must always be known statically. For instance, this `(rule ...)` stanza is rejected by Dune:

```
(rule (copy a b.%{read:file}))
```

### Directory targets

Note that at this time, Dune officially only supports user rules with targets in the current directory. Starting from Dune 3.24, Dune supports *directory targets*, where an action can produce a whole tree of build artifacts. To specify a directory target, you can use the `(dir <dirname>)` syntax. For example, the following stanza describes a rule with a file target `foo` and a directory target `bar`.

```
(rule
  (targets foo (dir bar))
  (action <action>))
```

In Dune 3.0 through 3.23, directory targets require the experimental `(using directory-targets 0.1)` extension.

### 4.1.15 alias

#### **(alias ...)**

Add dependencies to an alias.

Aliases do not need to be explicitly created, adding to a new name will “create” an alias. An alias with name `x` can be built by running `dune build @x`. See *Aliases*.

The common use of the `alias` stanza is make an alias depend on other ones:

```
(alias
  (name runtest)
  (deps
    (alias test-unit)
    (alias test-integration)))
```

#### **Warning**

In previous versions of the dune language, it was also possible to specify an action to run to construct the alias. Please use a *rule* stanza with the `alias` field instead:

```
(rule
  (alias benchmarks)
  (action (run %{exe:bench.exe})))
```

This stanza supports the following fields:

#### **(name <name>)**

An alias name.

Attaching dependencies to `(name x)` will ensure they are built by `dune build @x`.

This field is required.

#### **(deps <deps-conf list>)**

Specifies the dependencies of the alias.

See *Dependency Specification* for more details.

This field is required.

#### **(enabled\_if <blang expression>)**

Specifies the Boolean condition that must be true for the tests to run.

The condition is specified using the *Boolean Language*, and the field allows for *Variables* to appear in the expressions.

**(action <action>)**

Removed in version 2.0: use *rule* with the *alias* field instead.

An *action* for constructing the alias.

**(package <name>)**

Indicates that this alias stanza is part of package <name> and should be filtered out if <name> is filtered out from the command line, either with `--only-packages <pkgs>` or `-p <pkgs>`.

**(locks (<lock-names>))**

Specifies that the action must be run while holding the following locks. See *Locks* for more details.

### 4.1.16 copy\_files

The `copy_files` and `copy_files#` stanzas specify that files from another directory could be copied to the current directory, if needed.

The syntax is as follows:

```
(copy_files
 <optional-fields>
 (files <glob>))
```

<glob> represents the set of files to copy. See the *glob* for details.

<optional-fields> are:

- (alias <alias-name>) specifies an alias to which to attach the targets.
- (mode <mode>) specifies how to handle the targets. See *Modes* for details.
- (enabled\_if <blang expression>) conditionally disables this stanza. The condition is specified using the *Boolean Language*.
- (only\_sources <blang expression>) specifies that the glob in files gets applied over the source tree, and not the build tree.

The short form:

```
(copy_files <glob>)
```

is equivalent to:

```
(copy_files (files <glob>))
```

The difference between `copy_files` and `copy_files#` is the same as the difference between the `copy` and `copy#` actions. See *Actions* section for more details.

### 4.1.17 include

The `include` stanza allows including the contents of another file in the current dune file. The included file cannot be generated and must be present in the source tree. To include generated files, use the *dynamic\_include* stanza instead. This feature is intended for use in conjunction with promotion, when parts of a dune file are to be generated.

For instance:

```
(include dune.inc)

(rule (with-stdout-to dune.inc.gen (run ./gen-dune.exe)))

(rule
  (alias runtest)
  (action (diff dune.inc dune.inc.gen)))
```

With this dune file, running Dune as follows will replace the `dune.inc` file in the source tree by the generated one:

```
$ dune build @runtest --auto-promote
```

### 4.1.18 dynamic\_include

The `dynamic_include` stanza allows including the contents of another file in the current dune file like the `include` stanza. However, the `dynamic_include` stanza allows the included file to be the target of a rule and disallows generating some stanzas.

For instance:

```
(subdir b
  (dynamic_include ../a/foo.inc))
(subdir a
  (rule
    (write-file
      foo.inc
      "(rule (write-file file bar))"))))
```

In the example above, the dynamic rule loading and generation are split into different directories to avoid rule loading cycles as rules are loaded per directory.

The following stanzas cannot be dynamically generated:

- Libraries, Rocq theories, library redirects
- Public executables or install section with the `bin` section
- Plugin stanzas

### 4.1.19 env

The `env` stanza allows one to modify the environment. The syntax is as follows:

```
(env
  (<profile1> <settings1>)
  (<profile2> <settings2>)
  ...
  (<profilen> <settingsn>))
```

The first form (`<profile> <settings>`) that corresponds to the selected build profile will be used to modify the environment in this directory. You can use `_` to match any build profile.

Profiles are not declared by `env` stanzas. Any valid profile name can be selected on the command line or in `dune-workspace`. `env` stanzas only provide settings that are looked up later when a context is built with a matching profile. For example, a workspace context using (`profile dbg`) can pick up settings from an (`env (dbg ...)`) stanza in the project.

Fields supported in `<settings>` are:

- any OCaml flags field. See *OCaml Flags* for more details.
- `(link_flags <flags>)` specifies flags to OCaml when linking an executable. See *executables stanza*.
- `(c_flags <flags>)` and `(cxx_flags <flags>)` specify compilation flags for C and C++ stubs, respectively. See *library* for more details.
- `(env_vars (<var1> <val1>) .. (<varN> <valN>))` will add the corresponding variables to the environment where the build commands are executed and are used by `dune exec`. This field has been available since Dune 3.24. The dashed spelling `env-vars` is also supported for compatibility.
- `(menhir_flags <flags>)` specifies flags for Menhir stanzas. This flag was replaced by the `(menhir)` field (see below) starting in version 3.0 of the Menhir extension.
- `(menhir (flags <flags>) (explain <blang expression>))` specifies the Menhir settings. See *menhir* for more details. This field was introduced in version 3.0 of the Menhir extension.
- `(js_of_ocaml (flags <flags>)(build_runtime <flags>)(link_flags <flags>))` specifies `js_of_ocaml` flags. See *js\_of\_ocaml* for more details.
- `(js_of_ocaml (compilation_mode <mode>))` controls whether to use separate compilation or not where `<mode>` is either `whole_program` or `separate`.
- `(js_of_ocaml (sourcemap <mode>))` controls whether to generate sourcemap or not where `<mode>` is either `no`, `file` (to generate sourcemap in a `.map` file next to the generated javascript file) or `inline` (to inline the sourcemap at the end of the generated JavaScript file).
- `(js_of_ocaml (runtest_alias <alias-name>))` specifies the alias under which *Inline Tests* and tests (*tests*) run for the `js` mode.
- `(js_of_ocaml (enabled_if <blang expression>))` specifies whether the `js` mode is enabled. It is enabled by default.
- `(wasm_of_ocaml (flags <flags>)(build_runtime <flags>)(link_flags <flags>))` specifies `wasm_of_ocaml` flags. See *wasm\_of\_ocaml* for more details.
- `(wasm_of_ocaml (compilation_mode <mode>))` controls whether to use separate compilation or not where `<mode>` is either `whole_program` or `separate`.
- `(wasm_of_ocaml (sourcemap <mode>))` controls whether to generate sourcemap or not where `<mode>` is either `no`, `file` (to generate sourcemap in a `.map` file next to the generated javascript file) or `inline` (to inline the sourcemap at the end of the generated JavaScript file).
- `(wasm_of_ocaml (runtest_alias <alias-name>))` specifies the alias under which *Inline Tests* and tests (*tests*) run for the `wasm` mode.
- `(wasm_of_ocaml (enabled_if <blang expression>))` specifies whether the `wasm` mode is enabled. It is enabled by default.
- `(binaries <binaries>)`, where `<binaries>` is a list of entries of the form `(<filepath> as <name>)`. `(<filepath> as <name>)` makes the binary `<filepath>` available in the command search as just `<name>`. For instance, in a `(run <name> ...)` action, `<name>` will resolve to this file path. You can also write just the file path, in which case the name will be inferred from the basename of `<filepath>` by dropping the `.exe` suffix, if it exists. For example, `(binaries bin/foo.exe (bin/main.exe as bar))` would add the commands `foo` and `bar` to the search path.
- `(inline_tests <state>)`, where `<state>` is either `enabled`, `disabled`, or `ignored`. This field has been available since Dune 1.11. It controls the variable's value `%{inline_tests}`, which is read by the inline test framework. The default value is `disabled` for the release profile and `enabled` otherwise.
- `(odoc <fields>)` allows passing options to `odoc`. See *Passing Options to odoc* for more details.

- `(rocq <rocq_fields>)` allow passing options to Rocq. See *Rocq Environment Fields* for more details.
- `(formatting <settings>)` allows the user to set auto-formatting in the current directory subtree (see *formatting*).
- `(bin_annot <bool>)` allows the user to specify whether to generate `*.cmt` and `*.cmti` in the current directory subtree.

### 4.1.20 dirs

Added in version 1.6.

The `dirs` stanza allows specifying the subdirectories Dune will include in a build. The syntax is based on Dune's *Predicate Language* and allows the following operations:

- The special value `:standard` which refers to the default set of used directories. These are the directories that don't start with `.` or `_`.
- Set operations. Differences are expressed with backslash: `* \ bar`; unions are done by listing multiple items.
- Sets can be defined using globs.

Examples:

```
(dirs *) ;; include all directories
(dirs :standard \ ocaml) ;; include all dirs except ocaml
(dirs :standard \ test* foo*) ;; exclude all dirs that start with test or foo
```

Dune will not scan a directory that isn't included in this stanza. Any contained dune (or other special) files won't be interpreted either. It is still possible to depend on a specific file inside an excluded subdirectory by naming that file explicitly, but Dune won't discover files there by traversing the subdirectory.

Use *data\_only\_dirs* when Dune should ignore dune files inside a subdirectory but still treat the subdirectory's files as source files for dependency discovery. A data-only directory must still be included by `dirs`; if `dirs` excludes it, it is ignored rather than data-only.

#### Warning

Directory names should not contain any trailing slashes.

If you want to exclude a subdirectory, such as `foo/bar`, you need to use the *subdir* stanza:

```
(subdir foo (dirs :standard \ bar)) ;; exclude foo/bar
```

### 4.1.21 files

Added in version 3.21.

The `files` stanza allows restricting which files Dune should consider in the current directory. Its syntax mirrors the *Predicate Language* used by the `dirs` stanza and supports `:standard` (which expands to all files), globs, and set operations.

This is useful in mixed build setups where external tools such as `make` produce artifacts that Dune should ignore.

Examples:

```
(files :standard \ *.cm*) ;; ignore bytecode/native artifacts
(files *.ml *.mli) ;; only keep OCaml sources
```

### 4.1.22 data\_only\_dirs

Added in version 1.6.

Dune allows the user to treat directories as *data only*. `dune` files in these directories won't be evaluated for their rules, but the contents of these directories will still be usable as dependencies for other rules.

The syntax is a list of immediate subdirectory names or globs. Unlike the `dirs` stanza, this field doesn't support the full predicate language: `:standard`, `\`, and other set operations aren't available here.

`data_only_dirs` only changes the status of directories that Dune would otherwise include. If a directory is excluded by `dirs`, then Dune ignores it rather than treating it as data-only. For example, this marks `vendor` as ignored, not data-only, because it isn't included by `dirs`:

```
(dirs src)
(data_only_dirs vendor)
```

To make `vendor` data-only while excluding another generated directory, keep `vendor` in `dirs` and mark it with `data_only_dirs`:

```
(dirs :standard \ target)
(data_only_dirs vendor)
```

Example:

```
;; dune files in fixtures_* dirs are ignored
(data_only_dirs fixtures_*)
```

### 4.1.23 ignored\_subdirs

Deprecated since version 1.6.

One may also specify *data only* directories using the `ignored_subdirs` stanza, meaning it's the same as `data_only_dirs`, but the syntax isn't as flexible and only accepts a list of directory names. It's advised to switch to the new `data_only_dirs` stanza.

Example:

```
(ignored_subdirs (<sub-dir1> <sub-dir2> ...))
```

All of the specified `<sub-dirn>` will be ignored by Dune. Note that users should rely on the `dirs` stanza along with the appropriate set operations instead of this stanza. For example:

```
(dirs :standard \ <sub-dir1> <sub-dir2> ...)
```

### 4.1.24 include\_subdirs

The `include_subdirs` stanza is used to control how Dune considers subdirectories of the current directory. The syntax is as follows:

```
(include_subdirs <mode>)
```

Where `<mode>` maybe be one of:

- `no`, the default
- `unqualified`
- `qualified`

### Important

It's not allowed for a subdirectory of a directory with (`include_subdirs <x>`) (where `<x>` is not `no`) to contain one of the following stanzas:

- `library`
- `executable(s)`
- `test(s)`

### **no (the default)**

When the `include_subdirs` stanza isn't present or `<mode>` is `no`, Dune considers subdirectories independent.

### **unqualified**

When `<mode>` is `unqualified`, Dune will assume that the current directory's subdirectories are part of the same group of directories. In particular, Dune will simultaneously scan all these directories when looking for OCaml/Reason files. This allows you to split *library*, *executable* and *test* source files among several directories.

### Note

`unqualified` means that modules in subdirectories are seen as if they were all in the same directory. In particular, you cannot have two modules with the same name in two different directories.

### **qualified**

When `<mode>` is `qualified`, subdirectories are part of the module hierarchy. In the source tree, files in each subdirectory will be grouped into submodules of the *library*, *executable* or *test* module group, mirroring the directory structure.

Subdirectories are included recursively. However, recursion will stop when encountering a subdirectory that contains another `include_subdirs` stanza.

### Tip

The *ocamllex*, *ocamlyacc* and *menhir* stanzas must be defined in a dune file next to their corresponding source files, even when the directory group root is an ancestor.

## Module group interfaces

At each level of the source tree, Dune generates the module interface with aliases for all its sub-modules with a capitalized name of the directory.

- In `app.ml`, `sub/other.ml` is accessible at `Sub.Other`:

```
dune
app.ml
sub
└─ other.ml
```

Group interfaces are configurable similarly to the *library* stanza “library interface”.

- `sub/sub.ml` defines the module interface for the modules inside `sub/`:

```
dune
app.ml
sub
├─ sub.ml
└─ other.ml
```

### Warning

Currently *menhir* stanzas may not be used as a module group interface due to a [Dune issue](#).

## 4.1.25 vendored\_dirs

Added in version 1.11.

Dune supports vendoring other Dune-based projects natively, since simply copying a project into a subdirectory of your own project will work. Simply doing that has a few limitations though. You can work around those by explicitly marking such directories as containing vendored code.

Example:

```
(vendored_dirs vendor)
```

Dune will not resolve aliases in vendored directories. By default, it won't build all installable targets, run the tests, format, or lint the code located in such a directory while still building your project's dependencies. Libraries and executables in vendored directories will also be built with a `-w -a` flag to suppress all warnings and prevent pollution of your build output.

## 4.1.26 subdir

The `subdir` stanza can be used to evaluate stanzas in subdirectories. This is useful for generated files or to override stanzas in vendored directories without editing vendored dune files.

In this example, a `bar` target is created in the `foo` directory, and a `bar` target will be created in `a/b/bar`:

```
(subdir foo (rule (with-stdout-to bar (echo baz))))
(subdir a/b (rule (with-stdout-to bar (echo baz))))
```

## 4.1.27 cinaps

A `cinaps` stanza is available to support the `cinaps` tool. See the [cinaps website](#) for more details.

## 4.1.28 rocq.theory

See the documentation on the [rocq.theory](#), [rocq.extraction](#), [rocq.pp](#), and related stanzas.

## 4.1.29 mdx

Added in version 2.4.

MDX is a tool that helps you keep your markdown documentation up-to-date by checking that its code examples are correct. When setting an MDX stanza, the MDX checks are automatically attached to the `runtest` alias of the stanza's directory.

See [MDX's repository](#) for more details.

You can define an MDX stanza to specify which files you want checked.

Note that this feature is still experimental and needs to be enabled in your `dune-project` with the following `using` stanza:

```
(using mdx 0.4)
```

### Note

Version 0.2 of the stanza requires `mdx 1.9.0`. Version 0.4 of the stanza requires `mdx 2.3.0`.

The syntax is as follows:

```
(mdx <optional-fields>)
```

Where `<optional-fields>` are:

- `(files <globs>)` are the files that you want MDX to check, described as a list of globs (see the *Glob language specification*). It defaults to `*.md *.mld` as of version 0.4 of the stanza and `*.md` before.
- `(deps <deps-conf list>)` to specify the dependencies of your documentation code blocks. See *Dependency Specification* for more details.
- `(preludes <files>)` are the prelude files you want to pass to MDX. See *MDX's documentation* for more details on preludes.
- `(libraries <libraries>)` are libraries that should be statically linked in the MDX test executable.
- `(enabled_if <blang expression>)` is the same as the corresponding field of *library*.
- `(package <package>)` specifies which package to attach this stanza to (similarly to when `(package)` is attached to a `(rule)` stanza). When `-p` is passed, `(mdx)` stanzas with another package will be ignored. Note that this feature is completely separate from `(packages)`, which specifies some dependencies.
- `(locks <lock-names>)` specifies that the action of running the tests holds the specified locks. See *Locks* for more details.

### Upgrading from Version 0.1

- The 0.2 version of the stanza requires at least MDX 1.9.0. If you encounter an error such as, `ocaml-mdx: unknown command `dune-gen'`, then you should upgrade MDX.
- The field `(packages <packages>)` is deprecated in version 0.2. You can use `package` items in the generic `deps` field instead: `(deps (package <package>) ... (package <package>))`
- Use the new `libraries` field to directly link libraries in the test executable and remove the need for `#require` directives in your documentation code blocks.

### 4.1.30 menhir

A `menhir` stanza is available to support the Menhir parser generator.

To use Menhir in a Dune project, the language version should be selected in the `dune-project` file. For example:

```
(using menhir 3.0)
```

This will enable support for Menhir stanzas in the current project. If the language version is absent, Dune will automatically add this line with the latest Menhir version once a Menhir stanza is used anywhere.

The basic form for defining `menhir` parsers (analogous to *ocaml yacc*) is:

```
(menhir
 (modules <parser1> <parser2> ...)
 <optional-fields>)
```

<optional-fields> are:

- (merge\_into <base\_name>) is used to define modular parsers. This correspond to the --base command line option of menhir. With this option, a single parser named base\_name is generated.
- (flags <option1> <option2> ...) is used to pass extra flags to Menhir.
- (infer <bool>) is used to enable Menhir with type inference. This option is enabled by default with Menhir language 2.0.

Menhir supports writing the grammar and automation to the .cmly file. Therefore, if this is flag is passed to Menhir, Dune will know to introduce a .cmly target for the module.

- (explain <blang expression>) is used to control the generation of the .conflicts file explaining conflicts found while generating the parser. The condition is specified using the *Boolean Language*. This field was introduced in version 3.0 of the Menhir extension.

Note that starting in version 3.0 of the Menhir extension, the .conflicts file is generated by default. If this is not desired, it needs to be disabled explicitly by using the (explain) field.

### 4.1.31 ocamllex

(ocamllex <names>) is essentially a shorthand for:

```
(rule
 (target <name>.ml)
 (deps <name>.mll)
 (action (chdir %{workspace_root}
          (run %{bin:ocamllex} -q -o %{target} %{deps}))))))
```

To use a different rule mode, use the long form:

```
(ocamllex
 (modules <names>)
 (mode <mode>))
```

### 4.1.32 ocamlyacc

(ocamlyacc <names>) is essentially a shorthand for:

```
(rule
 (targets <name>.ml <name>.mli)
 (deps <name>.mly)
 (action (chdir %{workspace_root}
          (run %{bin:ocamlyacc} %{deps}))))))
```

To use a different rule mode, use the long form:

```
(ocamlyacc
 (modules <names>)
 (mode <mode>))
```

### 4.1.33 library\_parameter

**⚠ Warning**

This feature is experimental and requires the compiler you are using to support parameterised libraries.

The `library_parameter` stanza describes a parameter interface defined in a single `.mli` file. To enable this feature, you need to add `(using oxcaml 0.1) extension` in your `dune-project` file.

**(library\_parameter ...)**

Added in version 3.20.

Define a parameter.

**(name <parameter-name>)**

`parameter-name` is the name of the library parameter. It must be a valid OCaml module name as for *library*.

This must be specified if no `public_name` is specified.

**(public\_name ...)**

The name under which the library parameter can be referred as a dependency when it's not part of the current workspace, i.e., when it is installed. Without a `(public_name ...)` field, the library parameter won't be installed by Dune. The public name must start with the package name it's part of and optionally followed by a dot, then anything else you want. The package name must also be one of the packages that Dune knows about, as determined by the logic described in *Packages*.

**(package <package>)**

Installs a private library parameter under the specified package. Such a parameter is now usable by public libraries defined in the same project.

**(synopsis <string>)**

A one-line description of the library parameter.

**(modules <modules>)**

Specifies a specific module to select as a *library\_parameter*.

`<modules>` uses the *Ordered Set Language*, where elements are module names and don't need to start with an uppercase letter.

The library parameter **must** only declare one `mli` file as part of its modules.

**(libraries <library-dependencies>)**

Specifies the library parameter's dependencies.

See *Library Dependencies* for more details.

**(preprocess <preprocess-spec>)**

Specifies how to preprocess files when needed.

The default is `no_preprocessing`, and other options are described in *Preprocessing Specification*.

**(lint <preprocess-spec>)**

Specifies how to lint source files when building the `@lint` alias.

The default is `no_preprocessing`. The syntax is described in *Linting*.

**(preprocessor\_deps (<deps-conf list>))**

Specifies extra preprocessor dependencies preprocessor, i.e., if the preprocessor reads a generated file.

The specification of dependencies is described in *Dependency Specification*.

**(flags ...)**

See *OCaml Flags*.

**(ocamlc\_flags ...)**

See *OCaml Flags*.

**(ocamlopt\_flags ...)**

See *OCaml Flags*.

**(optional)**

If present, it indicates that the library parameter should only be built and installed if all the dependencies are available, either in the workspace or in the installed world.

**(enabled\_if <blang expression>)**

Conditionally disables a library parameter.

A disabled library parameter cannot be built and will not be installed.

The condition is specified using the *Boolean Language*, and the field allows for the `%{os_type}` variable, which is expanded to the type of OS being targeted by the current build. Its value is the same as the value of the `os_type` parameter in the output of `ocamlc -config`.

**(allow\_overlapping\_dependencies)**

Allows external dependencies to overlap with libraries that are present in the workspace.

### 4.1.34 jbuild\_version

Deprecated. This `jbuild_version` stanza is no longer used and will be removed in the future.

## 4.2 dune-project

Each `dune-project` file marks the root of a Dune projects within the current Dune workspace (see *Dune Projects and Workspaces*). It can also specify project-wide parameters.

The first line of `dune-project` must be a `lang` stanza with no extra whitespace or comments. The `lang` stanza controls the names and contents of all configuration files read by Dune and looks like:

```
(lang dune 3.14)
```

Additionally, they can contains the following stanzas.

### 4.2.1 accept\_alternative\_dune\_file\_name

**(accept\_alternative\_dune\_file\_name ...)**

Added in version 3.0.

Specify that the alternative filename `dune-file` is accepted in addition to `dune`.

This may be useful to avoid problems with `dune` files that have the executable permission in a directory in the `PATH`, which can unwittingly happen on Windows.

Note that `dune` continues to be accepted even after enabling this option, but if a file named `dune-file` is found in a directory, it will take precedence over `dune`.

## 4.2.2 authors

### (authors <strings>)

Added in version 1.9.

Specify authors.

Valid for all packages defined in the current Dune project. May be overridden by the per-package field (see *package*).

Example:

```
(authors
 "Jane Doe <jane.doe@example.com>"
 "John Doe <john.doe@example.com>")
```

## 4.2.3 bug\_reports

### (bug\_reports <url>)

Added in version 1.10.

Where bugs should be reported.

If a hosting service is used in (source), a default value is provided.

Valid for all packages defined in the current Dune project. May be overridden by the per-package field (see *package*).

Example:

```
(bug_reports https://dev.example.com/project/issues)
```

## 4.2.4 cram

### (cram <status>)

Define whether Cram-style tests are enabled for the project.

<status> can be either `enable` or `disable`. The default is `enable` starting from the language version 3.0.

 See also

*Cram Tests*

## 4.2.5 dialect

### (dialect ...)

Declare a new *dialect*.

#### (name <name>)

The name of the dialect being defined. It must be unique in a given project.

This field is required.

#### (implementation ...)

Details related to the implementation files (corresponding to *\*.ml*).

Changed in version 3.9: This field is made optional.

**(extension <string>)**

Specify the file extension used for this dialect.

The extension string must not start with a period and be unique in a given project (so that a given extension can be mapped back to a corresponding dialect). In Dune 3.9 and later, the extension string may contain periods (e.g., *cppo.ml*).

This field is required.

**(preprocess <action>)**

Run *<action>* to produce a valid OCaml abstract syntax tree.

This action is expected to read the file given in the variable named `%{input-file}` and output a *binary* abstract syntax tree on its standard output.

If the field is not present, it is assumed that the corresponding source code is already valid OCaml code and can be passed to the OCaml compiler as-is.

 **See also**

*Preprocessing With Actions*

**(format <action>)**

Run *<action>* to format source code for this dialect.

The action is expected to read the file given in the variable named `%{input-file}` and output the formatted source code on its standard output.

If the field is not present, the behavior depends on the presence of `(preprocess)`: if it is also not present (that is, the dialect consists of valid OCaml code), then the dialect will be formatted as any other OCaml code. Otherwise no special formatting will be done.

 **See also**

*How to Set up Automatic Formatting*

**(interface ...)**

Details related to the interface files (corresponding to *\*.mli*).

This field supports the same sub-fields as `implementation`.

Changed in version 3.9: This field is made optional.

**(merlin\_reader <program> <args>...)**

Configure Merlin to use *<program> <args>...* as `READER`. Merlin's `READER` is a mechanism to extend Merlin to support OCaml dialects by providing a program that transforms a dialect AST into an OCaml AST.

 **See also**

`merlin/src/extend/extend_protocol.ml` for the protocol specification.

This field is optional.

Added in version 3.16.

### 4.2.6 Default dialects

Dune ships with two dialects pre-configured and enabled:

- `ocaml` for the default OCaml syntax which consumes `.ml` and `.mli` files and uses `ocamlformat` for formatting.
- `reason` for the Reason syntax and enabled in `.rei/.rei` files. `refmt` is used for formatting.

A third dialect, `rescript`, is added when version 0.1 of Melange support (see *JavaScript Compilation With Melange*) is enabled in the project.

### 4.2.7 documentation

**(documentation <url>)**

Added in version 1.10.

Where the documentation is hosted.

Valid for all packages in the current Dune project. May be overridden by the per-package field (see *package*).

### 4.2.8 executables\_implicit\_empty\_intf

**(executables\_implicit\_empty\_intf ...)**

Added in version 2.9.

Automatically generate empty interface files for executables and tests that do not already have them.

By default, executables defined via `(executables(s) ...)` or `(test(s) ...)` stanzas are compiled with the interface file provided (e.g., `.mli` or `rei`). Since these modules cannot be used as library dependencies, it is common to give them empty interface files to strengthen the compiler's ability to detect unused values in these modules.

This option, when enabled, will generate an empty `*.mli` file.

Example:

```
(executables_implicit_empty_intf true)
```

This option is enabled by default starting with Dune lang 3.0.

### 4.2.9 expand\_aliases\_in\_sandbox

**(expand\_aliases\_in\_sandbox ...)**

When a sandboxed action depends on an alias, copy the expansion of the alias inside the sandbox. For instance, in the following example:

```
(alias
  (name foo)
  (deps ../x))

(cram
  (deps (alias foo)))
```

File `x` will be visible inside the Cram test if and only if this option is enabled. This option is a better default in general; however, it currently causes Cram tests to run noticeably slower. So it is disabled by default until the performance issue with Cram test is fixed.

## 4.2.10 explicit\_js\_mode

### (explicit\_js\_mode ...)

Do not implicitly add `js` to the `(modes ...)` field of executables.

In projects that use `dune lang 1.x`, JavaScript targets are defined for every bytecode executable. This is not very precise and does not interact well with the `@all` alias.

It is possible to opt out of this behavior by using:

```
(explicit_js_mode)
```

When this is enabled, an explicit `js` mode needs to be added to the `(modes ...)` field of executables in order to trigger the JavaScript compilation. Explicit JS targets declared like this will be attached to the `@all` alias.

Starting with Dune 2.0, this behavior is the default, and there is no way to disable it.

## 4.2.11 formatting

### (formatting ...)

Added in version 2.0.

Control automatic formatting. Several forms are accepted:

- To disable automatic formatting completely (equivalent to the behavior in language 1.x):

```
(formatting disabled)
```

- To restrict the languages that are considered for formatting:

```
(formatting
 (enabled_for <languages>))
```

The list of `<languages>` can be either `dune` (formatting of `dune` files) or a *dialect* name.

#### See also

[How to Set up Automatic Formatting](#)

## 4.2.12 generate\_opam\_files

### (generate\_opam\_files ...)

Added in version 1.10.

Use metadata specified in the `dune-project` file to generate `.opam` files.

To enable this integration, add the following field to the `dune-project` file:

```
(generate_opam_files)
```

#### See also

[How to Generate Opam Files from dune-project](#)

With this field, Dune can generate `*.opam` files from `dune-project` metadata. Build the `@opam` alias to refresh them explicitly:

```
$ dune build @opam
```

The `@install` and `@runtest` aliases also depend on `@opam`.

With (lang dune 3.23) or newer, updates to checked-in `*.opam` files are reported as promotions and must be applied with `dune promote`.

With older Dune language versions, generated `*.opam` files are promoted to the source tree automatically when these aliases are built.

To add opam fields not directly supported by `dune-project`, you can use `<package-name>.opam.template` files (see *Packages*).

### ➔ See also

*package* for fields supported by `dune-project`

*Packages* for more information on `opam.template` files

## 4.2.13 homepage

**(homepage <url>)**

Added in version 1.10.

The homepage of the project.

If a hosting service is used in `(source)`, a default value is provided.

Valid for all packages defined in the current Dune project. May be overridden by the per-package field (see *package*).

Example:

```
(bug_reports https://example.com/)
```

## 4.2.14 implicit\_transitive\_deps

**(implicit\_transitive\_deps <setting>)**

Control whether transitive dependencies are made implicitly visible during compilation.

`<setting>` is one of:

- `true` makes transitive dependencies implicitly visible. This is the default.
- `false` only listed dependencies are visible. If the `-H` flag is supported by the compiler (OCaml version  $\geq 5.2$ ) and Dune language version is  $\geq 1.17$ , Dune will pass the flag to the compiler, which avoids some corner cases (see below).
- `false-if-hidden-includes-supported` only listed dependencies are visible if the compiler supports the `-H` flag. Otherwise (OCaml version  $< 5.2$ ), the setting is ignored and all transitive dependencies are made visible. Introduced in Dune 3.20.

```
(implicit_transitive_deps false)
```

Then all dependencies directly used by a library or an executable must be added in the `libraries` field.

We recommend users experiment with this mode and report any problems.

Note that if `-H` flag is not being used, you must use `threads.posix` instead of `threads` when using this mode. This isn't an important limitation, as `threads.vm` is deprecated anyway.

In some situations, it can be desirable to selectively preserve the behavior of transitive dependencies' availability a library's users. For example, if we define a library `foo_more` that extends `foo`, we might want `foo_more` users to immediately have `foo` available as well. To do this, we must define the dependency on `foo` as re-exported:

```
(library
 (name foo_more)
 (libraries (re_export foo)))
```

#### 4.2.15 license

**(license <strings>)**

Added in version 1.9.

Specify the license of the project, ideally as an identifier from the [SPDX License List](#).

Valid for all packages defined in the current Dune project. May be overridden by the per-package field (see *package*).

Example:

```
(license MIT)
```

Multiple licenses may be specified.

#### 4.2.16 maintainers

**(maintainers <strings>)**

Added in version 1.10.

Specify maintainers.

Valid for all packages defined in the current Dune project. May be overridden by the per-package field (see *package*).

Example:

```
(maintainers
 "Jane Doe <jane.doe@example.com>"
 "John Doe <john.doe@example.com>")
```

#### 4.2.17 maintenance\_intent

**(maintenance\_intent <strings>)**

Added in version 3.18.

Specify the `opam` maintenance intent.

Valid for all packages defined in the current Dune project. May be overridden by the per-package field (see *package*).

Example:

```
(maintenance_intent "(latest)")
```

### 4.2.18 `map_workspace_root`

**(`map_workspace_root` <bool>)**

Control references to the file system locations where the project has been built.

- with (`map_workspace_root true`), dune rewrites references to the workspace root to `/workspace_root`. Note that when this mapping is enabled, the debug information produced by the bytecode compiler is incorrect, as the location information is lost.
- with (`map_workspace_root false`), the references are not rewritten.

The default is (`map_workspace_root true`).

Added in version 3.0: Initial version with the mapping always enabled.

Changed in version 3.7: Add a way to disable the mapping.

### 4.2.19 `name`

**(`name` <string>)**

Set the name of the project.

It is used by *dune subst* and error messages.

### 4.2.20 `opam_file_location`

**(`opam_file_location` <location>)**

Added in version 3.8.

Configure where generated `.opam` files are located. <location> can be one of the following:

- `relative_to_project`: the `.opam` files are generated in the project root directory. This is the default.
- `inside_opam_directory`: the `.opam` files are generated in a directory named `opam` in the project root directory.

 **See also**

*How to Generate Opam Files from dune-project*

### 4.2.21 `package`

This stanza is used to specify package metadata. In particular, this information is used when generating OPAM files (see *generate\_opam\_files*).

**(`package` ...)**

Define package-specific metadata.

**(`name` <string>)**

The name of the package.

This must be specified.

**(`synopsis` <string>)**

A short package description.

**(`description` <string>)**

A longer package description.

**(version <version>)**

Added in version 2.5.

The version of this package. This overrides the top-level *version* field for this package.

See *Package Version* for details on how Dune determines package versions.

**(depends <dep-specification>)**

Package dependencies, as *dep\_specification*.

**(conflicts <dep-specification>)**

Package conflicts, as *dep\_specification*.

**(depopts <dep-specification>)**

Optional package dependencies, as *dep\_specification*.

**(tags <tags>)**

A list of tags.

**(deprecated\_package\_names <name list>)**

A list of names that can be used with the *deprecated\_library\_name* stanza to migrate legacy libraries from other build systems that do not follow Dune's convention of prefixing the library's public name with the package name.

**(license ...)**

Added in version 2.0.

The same as (and takes precedences over) the corresponding global field (see *license*).

**(authors ...)**

Added in version 2.0.

The same as (and takes precedences over) the corresponding global field (see *authors*).

**(maintainers ...)**

Added in version 2.0.

The same as (and takes precedences over) the corresponding global field (see *maintainers*).

**(maintenance\_intent ...)**

Added in version 3.18.

The same as (and takes precedences over) the corresponding global field (see *maintenance\_intent*).

**(source ...)**

Added in version 2.0.

The same as (and takes precedences over) the corresponding global field (see *source*).

**(bug\_reports ...)**

Added in version 2.0.

The same as (and takes precedences over) the corresponding global field (see *bug\_reports*).

**(homepage ...)**

Added in version 2.0.

The same as (and takes precedences over) the corresponding global field (see *homepage*).

**(documentation ...)**

Added in version 2.0.

The same as (and takes precedences over) the corresponding global field (see *documentation*).

**(sites ...)**

Define a site.

(sites (<section> <name>) ...) defines a site named <name> in the section <section>.

**(allow\_empty)**

Added in version 3.0.

Allows packages that have no user-defined stanzas attached to them.

By default, starting from Dune 3.0, packages must contain at least one user-defined stanza (such as a *library*, *executable*, or *install* stanza). If a package is intentionally empty, add `(allow_empty)` to suppress the error.

Adding libraries to different packages is done via the `public_name` and `package` fields. See *library* section for details.

The list of dependencies `dep_specification` is modelled after opam's own language. The syntax is a list of the following elements:

```
op           ::=  '=' | '<' | '>' | '<>' | '>=' | '<='
filter      ::=  :dev | :build | :with-test | :with-doc | :with-dev-setup | :post
constr      ::=  (<op> <version>)
logop       ::=  or | and
dep         ::=  <name>
              (<name> <filter>)
              (<name> <constr>)
              (<name> (<logop> (<filter> | <constr>))*
dep_specification ::= <dep>+
```

Filters will expand to any opam variable name if prefixed by `:`, not just the ones listed in *filter*. This also applies to version numbers. For example, to generate depends: `[ pkg { = version } ],` use `(depends (pkg (= :version)))`.

Note that the use of a *using* stanza (see *using*) doesn't automatically add the associated library or tool as a dependency. They have to be added explicitly.

## 4.2.22 pin

** Warning**

*Dune Package Management* is not final yet and the configuration options are subject to change.

Pins are package overrides used in the context of package management. They allow to fix a package at a specific version which is not affected by the package repositories selected.

**(pin ...)**

Added in version 3.14.

Define a package override.

**(url <string>)**

The URL of the package source.

This can be a path to a directory on the local file system or remote Git repository. Local paths can be absolute or relative, and may optionally begin with `file://` though this is not necessary. Remote Git repository URLs must begin with `git+`, for example `git+https://github.com/user/repo` or `git+git@github.com:user/repo.git`.

This must be specified.

**(package ...)**

Defines which package is to be pinned.

This must be specified.

**(name <string>)**

The name of the package.

This must be specified.

**(version <string>)**

The version that the package should be assumed to be. Defaults to `dev` if unspecified.

**➔ See also**

*pin stanza in dune-workspace* for workspace-wide pinning.

## 4.2.23 source

**(source ...)**

Added in version 1.7.

Specify where the source for the package can be found.

It can be specified as `(uri <uri>)` or using shortcuts for some hosting services:

Valid for all packages defined in the current Dune project. May be overridden by the per-package field (see *package*).

Service	Syntax
GitHub	<code>(github user/repo)</code>
Bitbucket	<code>(bitbucket user/repo)</code>
GitLab	<code>(gitlab user/repo)</code> <code>(gitlab organization/project/repo)</code> ( <i>New in 3.17</i> )
SourceHut	<code>(sourcehut user/repo)</code>
Codeberg	<code>(codeberg user/repo)</code> ( <i>New in 3.17</i> )
Tangled	<code>(tangled @user.domain/repo)</code> ( <i>New in 3.21</i> )

Examples:

```
(source
 (github ocaml/dune))
```

```
(source
(uri https://dev.example.com/project.git))
```

#### 4.2.24 subst

**(subst <bool>)**

Added in version 3.0.

Control whether *dune subst* is enabled for this project.

- (subst disabled), means that any call of `dune subst` in this project is forbidden and will result in an error. This line will be omitted from the build instructions when generating opam files.
- (subst enabled) allows substitutions explicitly. This is the default.

#### 4.2.25 use\_standard\_c\_and\_cxx\_flags

**(use\_standard\_c\_and\_cxx\_flags ...)**

Added in version 2.8.

Control how flags coming from `ocamlc -config` are passed to the C compiler command line.

Historically, they have been systematically prepended without a way to override them.

If the following is passed, the mechanism is slightly altered:

```
(use_standard_c_and_cxx_flags)
```

In this mode, Dune will populate the `:standard` set of C flags with the content of `ocamlc_cflags` and `ocamlc_cppflags`. These flags can be completed or overridden using the *Ordered Set Language*.

This is the default in the language version 3.0.

 **See also**

*Flags in Foreign Code*

#### 4.2.26 using

**(using <plugin> <version>)**

Enable a dune language extension.

The language of configuration files read by Dune can be extended to support additional stanzas (e.g., `menhir`, `rocq.theory`, `mdx`).

`<plugin>` is the name of the plugin that defines this stanza and `<version>` describes the configuration language's version. Note that this version has nothing to do with the version of the associated tool or library. In particular, adding a `using` stanza will not result in a build dependency in the generated `.opam` file. See *generate\_opam\_files*.

Example:

```
(using mdx 0.3)
```

### 4.2.27 version

**(version <version>)**

Set the version of the project.

Example:

```
(version 1.2.3)
```

### 4.2.28 warnings

**(warnings ...)**

Added in version 3.11.

Configure Dune warnings for the project.

**(<name> <enabled | disabled>)**

Enable or disable the warning <name> for the current project.

### 4.2.29 wrapped\_executables

**(wrapped\_executables <bool>)**

Added in version 1.11.

Control wrapping of modules in executables.

Executables are made of compilation units whose names may collide with libraries' compilation units. To avoid this possibility, Dune prefixes these compilation unit names with `Dune__exe__`. This is entirely transparent to users except when such executables are debugged. In which case, the mangled names will be visible in the debugger.

- with `(wrapped_executables false)`, the original names are used.
- with `(wrapped_executables true)`, the names are mangled.

Starting in language version 2.0, the default value is `true`.

## 4.3 dune-workspace

A `dune-workspace` file (if present) marks the root of the current Dune workspace (see *Dune Projects and Workspaces*). It can be used to define compilation contexts (see *context*) and specify settings common to all Dune projects contained within the workspace.

By default, a workspace has only one build context named `default` which corresponds to the environment in which `dune` is run. You can define more contexts by writing a `dune-workspace` file.

You can point Dune to an explicit `dune-workspace` file with the `--workspace` option. For instance, it's good practice to write a `dune-workspace.dev` in your project with all the OCaml versions your projects support, so developers can test that the code builds with all OCaml versions by simply running:

```
$ dune build --workspace dune-workspace.dev @all @runtest
```

The `dune-workspace` file uses the S-expression syntax. This is what a typical `dune-workspace` file looks like:

```
(lang dune 3.14)
(context (opam (switch 4.08.1)))
(context (opam (switch 4.11.1)))
(context (opam (switch 4.14.2)))
```

The rest of this section describe the stanzas available.

Note that an empty `dune-workspace` file is interpreted the same as one containing exactly:

```
(lang dune 3.2)
(context default)
```

This allows you to use an empty `dune-workspace` file to mark the root of your project.

### 4.3.1 config stanzas

Starting in Dune 3.0, any of the stanzas from the `config` file can be used in the `dune-workspace` file. In this case, the configuration stanza will only affect the current workspace.

### 4.3.2 context

The `(context ...)` stanza declares a build context. Each context builds the same sources with a different configuration and stores its artefacts in a separate subdirectory of `_build/`. This is useful for cross compilation or building against multiple lock directories or opam switches.

By default, Dune uses a single context called `default` with artifacts in `_build/default/`. There are two kinds of context: `(default ...)` for a regular build context, or `(opam ...)` to use an opam switch.

#### Note

Despite its name, the `(default ...)` field of the `context` stanza specifies what we call the **regular context**, which is named this way to avoid confusion with the `default` context name.

```
(context
 (default
  <optional-fields>))
```

```
(context
 (opam
  (switch <opam-switch-name>)
  <optional-fields>))
```

<optional-fields> common to both context types are:

- `(name <name>)` is the name of the subdirectory within `_build` where this context's build artifacts will be stored. If omitted, the context name is `default` for the default context, and the switch name for the opam context.
- `(merlin)` instructs Dune to use this build context for Merlin.
- `(generate_merlin_rules)` instructs Dune to generate Merlin rules for this context, even if it is not the one selected via `(merlin)`.
- `(profile <profile>)` sets the profile for this *build context*. This takes precedence over the command-line option `--profile`.
- `(env <env>)` sets the environment for a particular context. This takes higher precedence over the root `env` stanza in the workspace file. This field has the same options as the `env` stanza.
- `(toolchain <findlib_toolchain>)` sets a `findlib` toolchain for the context.
- `(host <host_context>)` specifies a different context to build binaries that are meant to be executed on the host machine, such as preprocessors. See *Cross-Compilation* for more information. This is mutually exclusive with the `(targets ...)` field.

- `(targets <targets>)` specifies target names for cross compilation targets. See *Cross-Compilation* for more information. This is mutually exclusive with the `(host ...)` field.
- `(paths (<var1> <val1>) .. (<varN> <valN>))` allows you to set the value of any PATH-like variables in this context. If PATH itself is modified in this way, its value will be used to resolve workspace binaries, including finding the compiler and related tools. These variables will also be passed as part of the environment to any program launched by Dune. For each variable, the value is specified using the *Ordered Set Language*. Relative paths are interpreted with respect to the workspace root. See *Finding the Root*.
- `(fdo <target_exe>)` builds this context with feedback-directed optimizations. It requires OCamlFDO. `<target_exe>` is a path interpreted relative to the workspace root (see *Finding the Root*). `<target_exe>` specifies which executable to optimize. Users should define a different context for each target executable built with FDO. The context name is derived automatically from the default name and `<target_exe>`, unless explicitly specified using the `(name ...)` field. For example, if `<target_exe>` is `src/foo.exe` in a default context, then the name of the context is `default-fdo-foo` and the filename that contains execution counters is `src/fdo.exe.fdo-profile`. This feature is **experimental** and no backward compatibility is implied.
- `(instrument_with <instrumentation_backend>)` turns on instrumentation for the context. See *Instrumentation* for more information.
- `(disable_dynamically_linked_foreign_archives <bool>)` disables Dune's default behavior of building and installing dynamically-linked foreign archives (e.g., `dll*.so`), so bytecode executables are built with all foreign archives statically linked into the runtime system.

<optional-fields> specific to `(context (default ...))` are:

- **`(lock_dir <path>)` specifies the lock directory that will be used for building this context (if any).** If no lock directory is specified, `dune.lock` will be used. See the `lock_dir` stanza for lock directory configuration options.

<optional-fields> specific to `(context (opam ...))` are:

- **`(root <opam-root>)` is the opam root. By default, it will take the opam root defined by the environment in which dune is run, which is usually `~/opam`.**

Merlin reads compilation artifacts, and it can only read the compilation artifacts of a single context. Usually, you should use the artifacts from the `default` context, and if you have the `(context default)` stanza in your `dune-workspace` file, that is the one Dune will use.

For rare cases where this is not what you want, you can force Dune to use a different build context for Merlin by adding the field `(merlin)` to this context.

### 4.3.3 env

The `env` stanza can be used to set the base environment for all contexts in this workspace. This environment has the lowest precedence of all other `env` stanzas. The syntax for this stanza is the same as Dune's `env` stanza.

### 4.3.4 lock\_dir

#### Warning

*Dune Package Management* is not final yet and the configuration options are subject to change.

This stanza configures the lock directory settings for the current workspace. For the default workflow no configuration is necessary, but the defaults can be changed if desired.

### (lock\_dir ...)

Added in version 3.13.

Configures a specific lock directory to be created or used.

#### (path <string>)

The location in the source tree where the lock directory will be created or read from. If not specified defaults to `dune.lock`.

#### (repositories <name list>)

The repositories to be used for finding a package solution, specified in priority order. Supports `:standard` which contains `upstream` and `overlay`.

Additional repositories can be defined using the *repository* stanza.

#### (solver\_env ...)

The environment that is injected into the solver when creating the lock directory.

It consists of a sequence of (`<name>` `<value>`) pairs.

#### (unset\_variables <name list>)

A list of variables that are used in solving that are deliberately unset even if the solver could provide bindings for them.

The variables here cannot overlap with those defined in `solver_env`.

#### (pins <name list>)

Added in version 3.15.

Define which pins are enabled for this particular lock dir. See *pin* for details on how to define pins.

#### (version\_preference <string>)

Can be one of:

- **newest (default):** The solver will pick the newest available version of a package that satisfies the constraints.
- **oldest:** The solver will pick the lowest version that will satisfy the constraints

#### (constraints <dep-specification>)

Adds additional solver constraints that are passed to the solver. Follows the *dep\_specification* format.

#### Note

Names introduced through `constraints` are not considered dependencies and not added to the lockfile. They exist solely to add additional constraints if the packages to which the constraint is applied are selected and don't do anything otherwise.

#### (depopts <name list>)

Added in version 3.19.

Defines which optional package names (`depopts`) the solver should include when attempting to find a solution for the project.

### 4.3.5 pin

#### Warning

*Dune Package Management* is not final yet and the configuration options are subject to change.

This stanza is used to define additional package sources to use when locking a project and used for building dependencies.

#### Note

Defining a pin does not enable it by default. It needs to be enabled in a lock directory using the *lock\_dir* stanza.

#### (pin ...)

Added in version 3.15.

Defines a new package source.

##### (name <string>)

The name of the newly defined pin. This can be anything, it does not have to match the package.

This must be specified.

##### (url <string>)

This can be a path to a directory on the local file system or remote Git repository. Local paths can be absolute or relative, and may optionally begin with `file://` though this is not necessary. Remote Git repository URLs must begin with `git+`, for example `git+https://github.com/user/repo` or `git+git@github.com:user/repo.git`.

This must be specified.

##### (package ...)

Specifies the packages to assign this pin to.

##### (name <string>)

The name of the package.

This must be specified.

##### (version <string>)

The version that the package should be assumed to be. Defaults to dev if unspecified.

#### See also

*pin stanza in dune-project* for per-project pins.

### 4.3.6 pkg

#### Warning

*Dune Package Management* is not final yet and the configuration options are subject to change.

This stanza allows enabling/disabling the package management features of Dune.

### (pkg <setting>)

Added in version 3.20.

Where <setting> can be enabled or disabled.

Note that the command line option `--pkg` has precedence over this stanza.

### 4.3.7 profile

The build profile can be selected in the `dune-workspace` file by writing a `(profile ...)` stanza. For instance:

```
(profile release)
```

Note that the command line option `--profile` has precedence over this stanza.

### Profile Names and Environment Stanzas

Profiles do not need to be defined before they are selected. Any valid profile name can be passed to `--profile` or used in a workspace `(profile ...)` field. The selected profile name is then used to choose matching entries in *env* stanzas.

For example, this workspace context selects the profile `dbg`:

```
(context
 (default
  (name dbg)
  (profile dbg)))
```

A dune file in the project can provide settings for that profile with:

```
(env
 (dbg
  (flags (:standard -w +a))))
```

### Standard Profiles

The two standard profiles are `dev` and `release`.

`dev` is the default profile when none is selected explicitly. It is intended for interactive development and fast feedback. It enables development-oriented defaults such as treating OCaml warnings as errors and compiling local modules with `-opaque` when supported by the compiler. `-opaque` can make incremental rebuilds faster by reducing dependencies on implementation details, but it can also reduce cross-module optimization. The default value of `%{inline_tests}` is enabled in this profile.

`release` is the profile used for opam releases and by commands such as `dune build -p <package>`. It disables development-only defaults such as `warnings-as-errors` and `-opaque`. The default value of `%{inline_tests}` is disabled in this profile, so `inline-test` PPXs and backends can compile out test code unless an *env* stanza overrides it. This does not disable ordinary `test` or `tests` stanzas. Use `release` for artifacts that will be distributed or published.

User-defined profiles are not aliases for `dev` or `release`. They select matching *env* stanzas by name and otherwise use Dune's defaults for profiles that are neither `dev` nor `release`.

### Default OCaml Flags

For projects using Dune language 3.21 or later, Dune's default OCaml flags are:

- all profiles add `-g` to the default `ocamlc_flags` and `ocamlopt_flags`;
- the `dev` profile adds `-short-paths -keep-locs -warn-error +a` to the default common flags;

- release and user-defined profiles add no default common flags;
- no profile adds optimization flags such as `-O2` or `-O3` by default.

These defaults are the `:standard` set for the corresponding flag fields. They can be extended with `(:standard ...)` or replaced by omitting `:standard`. For example, `(ocamlc_flags ())` and `(ocamlopt_flags ())` remove the default `-g` flags.

For projects using older Dune language versions, the `dev` profile uses Dune's historical warning set. See the `%{dune-warnings}` variable in *Variables* for details.

### 4.3.8 repository

#### Warning

*Dune Package Management* is not final yet and the configuration options are subject to change.

This stanza defines a new named package repository and attaches a source location to it.

#### Note

Defining a repository does not enable it in project by default. It needs to be enabled in a lock directory using the `lock_dir` stanza.

#### **(repository ...)**

Added in version 3.12.

Defines a named package repository.

##### **(name <string>)**

The name used to refer to the repository. Names have to be unique.

This must be specified.

##### **(url <string>)**

The location from which the repository will be loaded.

Both HTTP and Git locations can be specified, the latter allowing for extensive control of the version by specifying an exact revision, tag or branch.

The URL scheme for Git locations need to specify the backend as `git` in the scheme part of the url, for instance:

```
git+https://github.com/ocaml/opam-repository.git
```

You can also use the SSH based URLs as follows:

```
git+ssh://git@github.com/ocaml/opam-repository
```

NOTE that the SSH URLs provided in the GitHub UI use a `:` to separate the repository owner and name from the hostname, but this needs to be replaced with a `/` to work correctly. `github.com:<username>/<repo>` needs to be changed to `github.com/<username>/<repo>`.

If the Git URLs need authentication, the user needs to set it up locally so `git` can authenticate and access these repositories.

This must be specified.

## 4.4 config

This file is used to set Dune's global configuration, which is applicable across projects and workspaces.

The configuration file is normally `~/.config/dune/config` on Unix systems and `%LOCALAPPDATA%/dune/config` on Windows. However, for most Dune commands, it is possible to specify an alternative configuration file with the `--config-file` option. Command-line flags take precedence over the contents of the `config` file. If `--no-config` or `-p` is passed, Dune will not read this file.

The `config` file can contain the following stanzas:

### 4.4.1 `action_stdout_on_success`

Specifies how Dune should handle the standard output of actions when they succeed. This can be used to reduce the noise of large builds.

```
(action_stdout_on_success <setting>)
```

where <setting> is one of:

- `print` prints the output on the terminal. This is the default.
- `swallow` ignores the output and does not print it on the terminal.
- `must-be-empty` enforces that the output should be empty. If it is not, Dune will fail.

### 4.4.2 `action_stderr_on_success`

Same as `action_stdout_on_success`, but applies to standard error instead of standard output.

### 4.4.3 `cache`

Specifies whether Dune is allowed to store and fetch build targets from the Dune cache.

```
(cache <setting>)
```

where <setting> is one of:

- `enabled` enables Dune cache.
- `enabled-except-user-rules` enables the Dune cache, but exclude user-written rules. This setting is a conservative choice that can avoid breaking rules whose dependencies are not correctly specified. Currently the default.
- `disabled` disables Dune cache.

### 4.4.4 `cache-check-probability`

While the main purpose of Dune cache is to speed up build times, it can also be used to check build reproducibility. It is possible to enable a probabilistic check, in which Dune will re-execute randomly chosen build rules and compare their results with those stored in the cache. If the results differ, the rule is not reproducible, and Dune will print out a corresponding warning.

```
(cache-check-probability <number>)
```

where <number> is a floating-point number between 0 and 1 (inclusive). 0 means never to check for reproducibility, and 1 means to always perform the check.

### 4.4.5 cache-storage-mode

Specify the mechanism used by the Dune cache for storage.

```
(cache-storage-mode <setting>)
```

where <setting> is one of:

- **auto** lets Dune decide the best mechanism to use.
- **hardlink** uses hard links for entries in the cache. If the cache is stored in a different partition than the one where the build is taking place, then this mode will not work and **copy** should be used instead.
- **copy** copies entries to the cache. This is less efficient than using hard links.

### 4.4.6 display

Specify the amount of Dune's verbosity.

```
(display <setting>)
```

where <setting> is one of:

- **progress**, Dune shows and updates a status line as build goals are being completed. This is the default value.
- **verbose** prints the full command lines of programs being executed by Dune, with some colors to help differentiate programs.
- **short** prints a line for each program executed with the binary name on the left and the targets of the action on the right.
- **quiet** only display errors.

### 4.4.7 jobs

Maximum number of concurrent jobs Dune is allowed to have.

```
(jobs <setting>)
```

where <setting> is one of:

- **auto**, auto-detect maximum number of cores. This is the default value.
- **<number>**, a positive integer specifying the maximum number of jobs Dune may use simultaneously.

This setting can also be controlled via the `-j` command-line option or the `DUNE_JOBS` environment variable. The command-line option takes precedence over the environment variable, which takes precedence over the configuration file.

### 4.4.8 pkg

Controls whether Dune's package management features are enabled. Package management includes dependency locking, package resolution, and building dependencies from source.

```
(pkg <setting>)
```

where <setting> is one of:

- **enabled** forces package management to be enabled, even if no lock directories are present.
- **disabled** forces package management to be disabled, even if lock directories are present. When disabled, Dune will not load package rules and all `dune pkg` commands will fail with an error.

If no `(pkg ...)` setting is specified, Dune will auto-detect whether to enable package management based on the presence of lock directories in the workspace. This usage is not currently recommended, since lock directories are not stabilized or portable.

Added in version 3.20.

### 4.4.9 `project_defaults`

Added in version 3.17.

Specify default values for stanzas `authors`, `maintainers`, and `license` of the *dune-project* file when initializing a project with `dune init proj`. The format of the `'project_defaults'` stanza is as follows:

```
(project_defaults
 <optional-fields>)
```

`<optional-fields>` are:

#### **(authors <string(s)>)**

Specify authors.

Example:

```
(project_defaults
 (authors
  "Jane Doe <jane.doe@example.com>"
  "John Doe <john.doe@example.com>"))
```

#### **(maintainers <string(s)>)**

Specify maintainers.

Example:

```
(project_defaults
 (maintainers
  "Jane Doe <jane.doe@example.com>"
  "John Doe <john.doe@example.com>"))
```

#### **(license <string(s)>)**

Specify license, ideally as an identifier from the [SPDX License List](#).

Example:

```
(project_defaults
 (license "MIT"))
```

### 4.4.10 `sandboxing_preference`

The preferred sandboxing setting. Individual rules may specify different preferences. Dune will try to utilize a setting satisfying both conditions.

```
(sandboxing_preference <setting> <setting> ...)
```

where each `<setting>` can be one of:

- `none` disables sandboxing.
- `hardlink` uses hard links for sandboxing. This is the default under Linux.

- `copy` copies files for sandboxing. This is the default under Windows.
- `symlink` uses symbolic links for sandboxing.

#### 4.4.11 terminal-persistence

Specifies how Dune handles the terminal when a rebuild is triggered in watch mode.

```
(terminal-persistence <setting>)
```

where <setting> is one of:

- `preserve` does not clear the terminal screen between rebuilds.
- `clear-on-rebuild` clears the terminal screen between rebuilds.
- `clear-on-rebuild-and-flush-history` clears the terminal between rebuilds, and it also deletes everything in the scrollback buffer.

## 4.5 Lexical Conventions

All configuration files read by Dune use a simple syntax that's similar to S-expressions. The Dune language can represent three kinds of values: atoms, strings, and lists. By combining these, it's possible to construct arbitrarily complex project descriptions.

A Dune configuration file is a sequence of atoms, strings, or lists separated by spaces, newlines, and comments. The other sections of this manual describe how each configuration file is interpreted, and we illustrate the syntax below:

### 4.5.1 Comments

The Dune language only has end of line comments. A semicolon introduces end of line comments and span up to the end of the current line. The system ignores everything from the semicolon to the end of the line. For instance:

```
; This is a comment
```

### 4.5.2 Atoms

An atom is a non-empty contiguous sequences of character other than special characters. Special characters are:

- spaces, horizontal tabs, newlines and form feed
- opening and closing parenthesis
- double quotes
- semicolons

For instance `hello` or `+` are valid atoms.

Note that backslashes inside atoms have no special meaning and Dune always interprets them as plain backslash characters.

### 4.5.3 Strings

A string is a sequence of characters surrounded by double quotes. A string represent the exact text between the double quotes, except for escape sequences. A backslash character introduces escape sequences. Dune recognizes and interprets the following escape sequences:

- `\n` to represent a newline character

- `\r` to represent a carriage return (character with ASCII code 13)
- `\b` to represent ASCII character 8
- `\t` to represent a horizontal tab
- `\NNN`, a backslash followed by three decimal characters to represent the character with ASCII code `NNN`
- `\xHH`, a backslash followed by two hexadecimal characters to represent the character with ASCII code `HH` in hexadecimal
- `\\`, a double backslash to represent a single backslash
- `\%{` to represent `%{` (see *Variables*)

Additionally, you can use a backslash just before the end of the line. This skips the newline leading up to the next non-space character. For instance, the following two strings represent the same text:

```
"abcdef"  
"abc\  
  def"
```

In most places where Dune expects a string, it will also accept an atom. As a result, it's possible to write most Dune configuration files using very few double quotes. This is very convenient in practice.

### 4.5.4 End of Line Strings

You can also write string using end of line strings. They are a convenient way to write blocks of text inside a Dune file.

The characters `"\|` or `"\>` introduce end of line strings and span to the end of the current line. If the next line also starts with `"\|` or `"\>`, Dune reads it as a continuation of the same string. For readability, either leave the text following the delimiter empty or start it with a space (that will be ignored).

For instance:

```
"\| this is a block  
"\| of text
```

represents the same text as the string `"this is a block\nof text"`.

Escape sequences are interpreted in text that follows `"\|` but not in text that follows `"\>`. Both delimiters can be mixed inside the same block of text.

### 4.5.5 Lists

Lists are sequences of values enclosed by parentheses. For instance `(x y z)` is a list containing the three atoms `x`, `y` and `z`. Lists can be empty, for instance: `()`.

Lists can be nested, allowing arbitrary representation for complex descriptions. For instance:

```
(html  
  (head (title "Hello world!"))  
  (body  
    This is a simple example of using S-expressions))
```

## 4.6 Actions

(`action ...`) fields describe user actions.

User actions are always run from the same subdirectory of the current build context as the dune file they are defined in, so for instance, an action defined in `src/foo/dune` will be run from `$build/<context>/src/foo`.

The argument of (`action ...`) fields is a small DSL that's interpreted by Dune directly and doesn't require an external shell. All atoms in the DSL support *Variables*. Moreover, you don't need to specify dependencies explicitly for the special `%{<kind>:...}` forms; these are recognized and automatically handled by Dune.

The DSL is currently quite limited, so if you want to do something complicated, it's recommended to write a small OCaml program and use the DSL to invoke it. You can use `shexp` to write portable scripts or *Configurator* for configuration related tasks. You can also use (*Experimental*) *Dune Action Plugin* to express program dependencies directly in the source code.

The following constructions are available:

### 4.6.1 run

(`run <prog> <args>`)

Execute a program. `<prog>` is resolved locally if Dune knows about a program by that name in the current workspace, otherwise it is resolved using the `PATH`.

“Available in the current workspace” does not mean that Dune searches the workspace source tree for an arbitrary executable file called `<prog>`. Instead, the program must be one of the binaries known to Dune's command lookup, for example:

- an executable installed by a workspace package, typically via an *executable* stanza with (`public_name ...`) or an *install* stanza with (`section bin`);
- a binary introduced by an *env* stanza's (`binaries ...`) field;
- or an executable found in `PATH`.

To run a private executable target from the source tree, refer to it by path, such as (`run ./tool.exe ...`). To make a private executable available by a bare command name, add it to the command lookup with an *env* stanza:

```
(env
  (
    (binaries
      (./tool.exe as tool))))
```

Example:

```
(run capnp compile -o %{bin:capnpc-ocaml} schema.capnp)
```

### 4.6.2 system

(`system <cmd>`)

Execute a command using the system shell: `sh` on Unix and `cmd` on Windows.

Example:

```
(system "command arg1 arg2")
```

### 4.6.3 bash

**(bash <cmd>)**

Execute a command using `/bin/bash`. This is obviously not very portable.

Example:

```
(bash "echo $PATH")
```

### 4.6.4 dynamic-run

**(dynamic-run <prog> <args>)**

Execute a program that was linked against the `dune-action-plugin` library. `<prog>` is resolved in the same way as in `run`.

Example:

```
(dynamic-run ./plugin.exe)
```

### 4.6.5 chdir

**(chdir <dir> <DSL>)**

Run an action in a different directory.

Example:

```
(chdir src
 (run ./build.exe))
```

### 4.6.6 setenv

**(setenv <var> <value> <DSL>)**

Run an action with an environment variable set.

Example:

```
(setenv
  VAR value
  (bash "echo $VAR"))
```

### 4.6.7 with-accepted-exit-codes

**(with-accepted-exit-codes <pred> <DSL>)**

Added in version 2.0.

Specifies the list of expected exit codes for the programs executed in `<DSL>`. `<pred>` is a predicate on integer values, and it's specified using the *Predicate Language*. `<DSL>` can only contain nested occurrences of `run`, `bash`, `system`, `chdir`, `setenv`, `ignore-<outputs>`, `with-stdin-from`, and `with-<outputs>-to`.

Example:

```
(with-accepted-exit-codes
 (or 1 2)
 (run false))
```

### 4.6.8 echo

**(echo <string>)**

Output a string on stdout.

Example:

```
(echo "Hello, world")
```

### 4.6.9 with-<outputs>-to

**(with-<outputs>-to <file> <DSL>)**

Redirect the output to a file, where <outputs> is one of: stdout, stderr or outputs (for both stdout and stderr).

Example:

```
(with-stdout-to conf.txt  
(run ./get-conf.exe))
```

### 4.6.10 with-stdin-from

**(with-stdin-from <file> <DSL>)**

Redirect the input from a file.

Example:

```
(with-stdin-from data.txt  
(run ./tests.exe))
```

### 4.6.11 ignore-<outputs>

**(ignore-<outputs> <DSL>)**

Ignore the output, where <outputs> is one of: stdout, stderr, or outputs.

Example:

```
(ignore-stderr  
(run ./get-conf.exe))
```

### 4.6.12 cat

**(cat <file> ...)**

Sequentially print the contents of files to stdout.

Example:

```
(cat data.txt)
```

### 4.6.13 copy

### (copy <src> <dst>)

Copy a file. If these files are OCaml sources, you should follow the `module_name.xxx.ml` *naming convention* to preserve Merlin's functionality.

Example:

```
(copy data.txt.template data.txt)
```

### 4.6.14 copy#

#### (copy# <src> <dst>)

Copy a file and add a line directive at the beginning.

Example:

```
(copy# config.windows.ml config.ml)
```

More precisely, `copy#` inserts the following line:

```
# 1 "<source file name>"
```

Most languages recognize such lines and update their current location to report errors in the original file rather than the copy. This is important because the copy exists only under the `_build` directory, and in order for editors to jump to errors when parsing the build system's output, errors must point to files that exist in the source tree. In the beta versions of Dune, `copy#` was called `copy-and-add-line-directive`. However, most of time, one wants this behavior rather than a bare copy, so it was renamed to something shorter.

### 4.6.15 write-file

#### (write-file <file> <string>)

Writes `<string>` to `<file>`.

Example:

```
(write-file users.txt jane,joe)
```

### 4.6.16 pipe-<outputs>

#### (pipe-<outputs> <DSL> <DSL> <DSL>...)

Added in version 2.7.

Execute several actions (at least two) in sequence, filtering the `<outputs>` of the first command through the other command, piping the standard output of each one into the input of the next.

Example:

```
(pipe-stdout  
(run ./list-tests.exe)  
(run ./exec-tests.exe))
```

### 4.6.17 format-dune-file

**(format-dune-file <src> <dst>)**

Added in version 3.18.

Output the formatted contents of the file <src> to <dst>. The source file is assumed to contain S-expressions. Note that the precise formatting can depend on the version of the Dune language used by containing project.

Example:

```
(format-dune-file file.sexp file.sexp.formatted)
```

**4.6.18 diff****(diff <file1> <file2>)**

(diff <file1> <file2>) is similar to (run diff <file1> <file2>) but is better and allows promotion. See *Diffing and Promotion* for more details.

Example:

```
(diff test.expected test.output)
```

**4.6.19 diff?****(diff? <file1> <file2>)**

(diff? <file1> <file2>) is similar to (diff <file1> <file2>) except that <file2> should be produced by a part of the same action rather than be a dependency, is optional and will be consumed by diff?.

Example:

```
(progn
  (with-stdout-to test.output (run ./test.exe))
  (diff? test.expected test.output))
```

**4.6.20 cmp****(cmp <file1> <file2>)**

(cmp <file1> <file2>) is similar to (run cmp <file1> <file2>) but allows promotion. See *Diffing and Promotion* for more details.

Example:

```
(cmp bin.expected bin.output)
```

**4.6.21 progn****(progn <DSL> ...)**

Execute several commands in sequence.

Example:

```
(progn
  (run ./proga.exe)
  (run ./progb.exe))
```

### 4.6.22 concurrent

**(concurrent <DSL> ...)**

Execute several commands concurrently and collect all resulting errors, if any.

**⚠ Warning**

The concurrency is limited by the `-j` flag passed to Dune. In particular, if Dune is running with `-j 1`, these commands will actually run sequentially, which may cause a deadlock if they talk to each other.

Example:

```
(concurrent
 (run ./proga.exe)
 (run ./progb.exe))
```

### 4.6.23 no-infer

**(no-infer <DSL>)**

Perform an action without inference of dependencies and targets. This is useful if you are generating dependencies in a way that Dune doesn't know about, for instance by calling an external build system.

Example:

```
(no-infer
 (progn
  (run make)
  (copy mylib.a lib.a)))
```

Note: expansion of the special `%{<kind>:...}` is done relative to the current working directory of the DSL being executed. So for instance, if you have this action in a `src/foo/dune`:

```
(action (chdir ../../.. (echo %{dep:dune})))
```

Then `%{dep:dune}` will expand to `src/foo/dune`. When you run various tools, they often use the filename given on the command line in error messages. As a result, if you execute the command from the original directory, it will only see the basename.

To understand why this is important, let's consider this dune file living in `src/foo`:

```
(rule
 (target blah.ml)
 (deps blah.mll)
 (action
  (run ocamllex -o %{target} %{deps})))
```

Here the command that will be executed is:

```
$ ocamllex -o blah.ml blah.mll
```

And it will be executed in `_build/<context>/src/foo`. As a result, if there is an error in the generated `blah.ml` file, it will be reported as:

```
File "blah.ml", line 42, characters 5-10:
Error: ...
```

Which can be a problem, as your editor might think that `blah.ml` is at the root of your project. Instead, this is a better way to write it:

```
(rule
 (target blah.ml)
 (deps blah.ml)
 (action
  (chdir %{workspace_root}
   (run ocamllex -o %{target} %{deps}))))
```

## 4.7 Ordered Set Language

A few fields take an ordered set as argument and can be specified using a small DSL.

Dune interprets this DSL into an ordered set of strings using the following rules:

- `:standard` denotes the standard value of the field when it's absent
- an atom not starting with a `:` is a singleton containing only this atom
- a list of sets is the concatenation of its inner sets
- `(<sets1> \ <sets2>)` is the set composed of elements of `<sets1>` that do not appear in `<sets2>`

In addition, some fields support the inclusion of an external file using the syntax `(:include <filename>)`. For instance, this is useful when you need to run a script to figure out some compilation flags. `<filename>` is expected to contain a single S-expression and cannot contain `(:include ...)` forms.

Note that inside an ordered set, a list's first element cannot be an atom except if it starts with `-` or `:`. The reason for this is that we're planning to add simple programmatic features in the future so that one may write:

```
(flags (if (>= %{ocaml_version} 4.08) ...))
```

This restriction will allow you to add this feature without introducing breaking changes. If you want to write a list where the first element doesn't start with `-`, you can simply quote it: `("x" y z)`.

Most fields using the ordered set language also support *Variables*. Variables are expanded after the set language is interpreted.

## 4.8 Boolean Language

The Boolean language allows the user to define simple Boolean expressions that Dune can evaluate. Here's a semiformal specification of the language:

```
op ::= '=' | '<' | '>' | '<>' | '>=' | '<='
expr ::= (and <expr>+)
        (or <expr>+)
        (<op> <template> <template>)
        (not <expr>)
        <template>
```

After an expression is evaluated, it must be exactly the string `true` or `false` to be considered as a Boolean. Any other value will be treated as an error.

Below is a simple example of a condition expressing that the build has a Flambda compiler, with the help of variable expansion, and is targeting OSX:

```
(and %{ocaml-config:flambda} (= %{ocaml-config:system} macosx))
```

## 4.9 Predicate Language

The predicate language allows the user to define simple predicates (Boolean-valued functions) that Dune can evaluate. Here is a semiformal specification of the predicate language:

```
pred ::= (and pred pred)
       (or  pred pred)
       (not pred)
       :standard
       element
```

The exact meaning of `:standard` and the nature of `element` depend on the context. For example, in the case of the *subdir*, an `element` corresponds to file glob patterns. Another example is the user action *with-accepted-exit-codes*, where an `element` corresponds to a literal integer.

## 4.10 Library Dependencies

Library dependencies are specified using `(libraries ...)` fields in `library` and `executables` stanzas.

For libraries defined in the current scope, you can either use the real name or the public name. For libraries that are part of the *installed world*, or for libraries that are part of the current workspace but in another scope, you need to use the public name. For instance: `(libraries base re)`.

When resolving libraries, ones that are part of the workspace are always preferred to ones that are part of the *installed world*.

### 4.10.1 Alternative Dependencies

Sometimes, one doesn't want to depend on a specific library but rather on whatever is already installed, e.g., to use a different backend, depending on the target.

Dune allows this by using a `(select ... from ...)` form inside the list of library dependencies.

Select forms are specified as follows:

```
(select <target-filename> from
 (<literals> -> <filename>)
 (<literals> -> <filename>)
 ...)
```

`<literals>` are lists of literals, where each literal is one of:

- `<library-name>`, which will evaluate to true if `<library-name>` is available, either in the workspace or in the *installed world*
- `!<library-name>`, which will evaluate to true if `<library-name>` is not available in the workspace or in the *installed world*

When evaluating a select form, Dune will create `<target-filename>` by copying the file given by the first (`< literals> -> <filename>`) case where all the literals evaluate to true. It is an error if none of the clauses are selectable. You can add a fallback by adding a clause of the form (`-> <file>`) at the end of the list.

### 4.10.2 Re-Exported Dependencies

A dependency `foo` may be marked as always *re-exported* using the following syntax:

```
(re_export foo)
```

For instance:

```
(library
 (name bar)
 (libraries (re_export foo)))
```

This states that this library explicitly re-exports the interface of `foo`. Concretely, when something depends on `bar`, it will also be able to see `foo` independently of whether *implicit transitive dependencies* are allowed or not. When they are allowed, which is the default, all transitive dependencies are visible, whether they are marked as re-exported or not.

### 4.10.3 Instantiating Parameterised Dependencies

This feature requires OxCaml, see *library\_parameter*.

A parameterised dependency `foo` can be instantiated with the arguments `bar`, `qux` using the syntax:

```
(instantiate foo bar qux)
```

For example:

```
(library
 (name test)
 (libraries (instantiate foo bar qux)))
```

The library `foo` must have declared the set of parameters it expects, and the arguments given to the instantiation must implement a subset of these parameters. The ordering of the arguments does not matter, as the instantiation relies on the implemented parameter to uniquely identify each argument. For executables, the parameterised dependencies must be fully instantiated.

In the OCaml code, the instantiated library will be available under the module name `Foo`. To avoid overlapping module names when instantiating the same dependency multiple times, the syntax `:as` allows renaming the module. For example:

```
(library
 (name test)
 (libraries
 (instantiate foo a b :as foo_a_b)
 (instantiate foo bar qux :as foo_bar_qux)))
```

Then the instantiations will be available under the names `Foo_a_b` and `Foo_bar_qux`.

Dependencies automatically inherit the parameters of their parent library. For example, assuming the parameterised library `foo` requires two parameters `p` and `q`:

```
(library
 (name test)
```

(continues on next page)

(continued from previous page)

```
(parameters p q)
(libraries
  (instantiate foo :as foo_implicit)
  (instantiate foo an_implementation_of_q :as foo_q)
  (instantiate foo bar qux :as foo_bar_qux)
  other_foo))
```

Then `foo_implicit` is implicitly `(instantiate foo p q)`, while `(instantiate foo an_implementation_of_q)` will only inherit the parameter `p`.

If `other_foo`, which is not explicitly instantiated here, is also parameterised by the parameters `p` (and) or `q`, it will also inherit its parent arguments. Dune will report an error if a dependency requires parameters which have neither been given explicitly given via an instantiation and are not listed in the parent library parameters.

For unwrapped libraries, the instantiation of parameterised libraries is not currently generated. This is subject to change soon, but in the mean time, you'll need to manually declare the instantiations: If you depend on the instantiation `(instantiate foo bar qux :as new_name)` with `bar` an implementation of the parameter `param_bar` and `qux` an implementation of `param_qux`, then you'll need to write the following:

```
module New_name = Foo (Param_bar) (Bar) (Param_qux) (Qux) [:@jane.non_erasable.instances]
```

### **Note**

While this reuses the OCaml functor application syntax, the attribute changes the meaning: The `(Param)` `(Impl)` must go together as a pair, but the ordering of the arguments otherwise does not matter.

## 4.11 Preprocessing Specification

Some stanzas including `(library)` accept a `(preprocess)` field. The possible values for its argument are:

```
pp-spec      ::= <pp-module>
              (per-module <per-module>+)
per-module   ::= ( <pp-module> <module>+ )
pp-module    ::= no_preprocessing
              (action <action>)
              (pps <ppx-rewriters-and-flags>)
              (staged_pps <ppx-rewriters-and-flags>)
              future_syntax
```

### 4.11.1 no\_preprocessing

When `no_preprocessing` is passed, files are given as-is to the compiler. This is the default behavior.

### 4.11.2 Preprocessing With Actions

In `(action <action>)`, `<action>` uses the same DSL as described in *Actions*, and for the same reason given in that section, it will be executed from the root of the current build context. It's expected to be an action that reads the file given as a dependency named `input-file` and outputs the preprocessed file on its standard output.

More precisely, `(preprocess (action <action>))` acts as if you had set up a rule for every file of the form:

```
(rule
  (target file.pp.ml)
  (deps file.ml)
  (action
    (with-stdout-to %{target}
      (chdir %{workspace_root} <action>))))
```

The equivalent of a `-pp <command>` option passed to the OCaml compiler is `(system "<command> %{input-file}")`.

### 4.11.3 Using PPX Rewriters

If `(pps <ppx-rewriters-and-flags>)` is used, the corresponding rewriters are set up using the “fast pipeline” (using a separate preprocessing step). If `(staged_pps <ppx-rewriters-and-flags>)` is used, they are set up using the “classic pipeline” (using the `-ppx` command-line argument).

The distinction between these pipelines is explained in *How Preprocessing Works*.

PPX rewriters need to be compiled as a driver to be used by Dune. To run PPXs that do not support this (usually old ones), it is possible to use the `ppxfind` tool.

### 4.11.4 Arguments to PPX Rewriters

In `(pps <ppx-rewriters-and-flags>)` and `(staged_pps <ppx-rewriters-and-flags>)`, `<ppx-rewriters-and-flags>` is a sequence where each element is either a command line flag if it starts with a `-` or the name of a library.

If you want to pass command line flags that don’t start with a `-`, you can separate library names from flags using `--`. So for instance from the following preprocess field:

```
(preprocess (pps ppx1 -foo ppx2 -- -bar 42))
```

The list of libraries will be `ppx1` and `ppx2`, and the command line arguments will be: `-foo -bar 42`.

### 4.11.5 Future Syntax

The `future_syntax` specification is a special value that brings some of the newer OCaml syntaxes to older compilers.

It is equivalent to `no_preprocessing` when using one of the most recent versions of the compiler. When using an older one, it is a shim preprocessor that backports some of the newer syntax elements. This allows you to use some of the new OCaml features while keeping compatibility with older compilers.

One example of supported syntax is the custom `let-syntax` that was introduced in 4.08, allowing the user to define custom `let` operators.

Note that this feature is implemented by the third-party `ocaml-syntax-shims` project, so if you use this feature, you must also declare a dependency on this package.

### 4.11.6 Per-Module Preprocessing Specification

By default, a preprocessing specification applies to all modules in the library/set of executables. It’s possible to select the preprocessing on a module-by-module basis by using the `(per-module ...)` syntax. For instance:

```
(preprocess
  (per_module
    ((action (run ./pp.sh X=1 %{input-file})) foo bar)
    ((action (run ./pp.sh X=2 %{input-file})) baz)))
```

The modules `Foo` and `Bar` will be preprocessed with `pp.sh X=1`, and `Baz` will be preprocessed with `pp.sh X=2`.

### 4.11.7 Preprocessor Dependencies

If your preprocessor needs extra dependencies, you should use the `preprocessor_deps` field available in the `library`, `executable`, and `executables` stanzas. It uses the *Dependency Specification* to declare what the preprocessor needs.

### 4.11.8 Linting

The `lint` field available in `library`, `executable`, `executables`, `test`, `tests`, `library_parameter`, and `melange.emit` stanzas uses syntax similar to `preprocess`:

```
(lint <preprocess-spec>)
```

The default is `no_preprocessing`. Linting actions are attached to the `@lint` alias and are run by building it, for example with `dune build @lint`.

The `future_syntax` and `staged_pps` forms are not supported in `lint` fields. The `(per_module ...)` form is supported, as with `preprocess`.

With `(lint (action <action>))`, Dune runs `<action>` for each matching implementation and interface source. The `%{input-file}` variable is bound to the source file, as for preprocessing actions. The action should report lint errors in the usual way, such as by exiting with a non-zero status.

With `(lint (pps ...))`, Dune runs the PPX driver in lint mode. Corrections registered by the PPX, for example through `ppplib`, are displayed as diffs against the source files and can be applied with `dune promote`.

## 4.12 Cram Tests

### 4.12.1 Synopsis

Cram tests are integrations tests that describe a shell session. These tests contain commands and expected outputs. When executed, the commands are executed and the actual output is compared to the expected output.

Here is an example showing how `echo`, `cat`, and `rm` interact.

*Create a file:*

```
$ echo contents > data.txt
```

*Display it:*

```
$ cat data.txt
contents
```

*Remove it:*

```
$ rm data.txt
```

*Try to remove it again:*

```
$ rm data.txt
rm: cannot remove 'data.txt': No such file or directory
[1]
```

The syntax mimics a shell session: there are comments and shell commands with their output.

## 4.12.2 Examples

### Simple Commands

This is the simplest test case: it executes the command `touch this-file.txt` and expects that the command has no output.

```
$ touch this-file.txt
```

### Output

This executes `ls` and expects it to display `this-file.txt`:

```
$ ls
this-file.txt
```

There can be several output lines if the command is expected to print several lines. Also, note that if a command has no output, the next one can come in the next line.

```
$ touch other-file.txt
$ ls
other-file.txt
this-file.txt
```

### Comments

Lines that are not indented are ignored. These act as comments.

```
"touch" will create an empty file:
```

```
  $ touch data.txt
```

```
Printing it will do nothing:
```

```
  $ cat data.txt
```

### Continuation Lines

Continuation lines are used when a command fits on several lines. This can happen in all the cases where pressing Enter would not run the command. For example, when passing a backslash character to escape the line ending. In that case, all the continuation lines are grouped together as a single command.

This syntax mimics the PS2 prompt in shells - the “>” character is not passed to the command.

```
$ echo \  
> a \  
> b \  
> d \  
> c  
a b c d
```

This is often used with shell “heredocs” to create files:

```
$ cat > file.txt << EOF  
> Everything
```

(continues on next page)

(continued from previous page)

```
> here will
> written to
> the file
> EOF

$ cat file.txt
Everything
here will
written to
the file
```

## Exit Codes

When a command exits with a nonzero exit code, it is displayed between square brackets after its output:

```
$ false
[1]

$ echo hello; false
hello
[1]
```

### 4.12.3 Syntax Details

Cram tests are parsed line by line, depending on the first characters of each line:

- If a line starts with `␣$␣` (`␣` denoting a space character), the rest is a command.
- If it starts with `␣>␣`, the rest is the continuation of a command (continuation lines must immediately follow a command).
- If it start with `␣` and something else, the rest is the expected output or exit code of the previous command.
- Everything else is a comment.

### 4.12.4 File and Directory Tests

There are two types of Cram tests: file tests and directory tests. File tests are files with a `.t` extension. Directory tests are files named `run.t` within a directory with a name that ends with `.t`.

A Cram test begins its execution in a temporary directory where its dependencies (as listed in the corresponding *cram stanzas*, if any) are available. In the case of a directory test, the contents of the directory are also available.

File tests have the nice property that they are self-contained: everything happens in a single file. This is handy because it does not make a deep file hierarchy in a project. But if the test requires some files, these need to be created using `cat` and heredocs. Directory tests, on the other hand, allow creating these test fixtures as normal files. This can be more comfortable because it makes the usual tooling (syntax highlighting, completion, etc.) available.

### 4.12.5 Executing Cram Tests

Every Cram test has a name. For file tests, the name of `something.t` is `something`, and for directory tests, the name of `something.t/run.t` is `something`.

There are several ways to execute Cram tests:

- Running `dune runtest something.t` will run the cram test with filename `something.t`.

- All Cram tests are attached to the `@runtest` alias. So `dune runtest` will run all Cram tests.
- Every Cram test creates an alias after its name. So, `dune build @something` will run tests named `something`.

When a Cram test is executed, the commands it contains are executed, and a corrected file is created where the command outputs are inserted after each command. This corrected file is then offered for *promotion* by Dune.

Concretely, this means that Dune will display the difference between the Cram test's current contents and the latest run's output. This diff can be applied by running `dune promote`, as usual.

```
$ touch changed-name.txt
$ ls
-other-file.txt
+changed-name.txt
this-file.txt
```

## Early Shell Exits

If a command causes the shell to exit early, all subsequent commands become unreachable. Dune marks these with `***** UNREACHABLE *****`:

```
Commands after an early exit are unreachable:

$ exit 1
***** UNREACHABLE *****
$ echo "this won't run"
***** UNREACHABLE *****
```

Output from commands that did execute can still be promoted normally.

A common cause of unexpected early exits is the POSIX special builtin behavior. Cram tests run with `sh` by default, which operates in POSIX mode. In POSIX, `.` (source) is a *special builtin*, and when a special builtin fails, a non-interactive shell exits immediately.

This means `source foo.sh` (without `./`) can cause the shell to exit if `foo.sh` is not found in `PATH`:

```
This will exit the shell because foo.sh is not in PATH:

$ source foo.sh
***** UNREACHABLE *****

Use an explicit path to avoid this:

$ source ./foo.sh
```

## 4.13 Variables

Some fields can contain variables that are expanded by Dune. The syntax of variables is as follows:

```
{var}
```

or, for more complex forms that take an argument:

```
{fun:arg}
```

In order to write a plain `%{`, you need to write `\%{` in a string.

Unless otherwise noted, path variables in actions refer to paths in the current build context and are rendered relative to the action's current working directory. By default, an action in `src/foo/dune` runs from `_build/default/src/foo`, so `%{workspace_root}` may be rendered as `../..`. These variables do not point directly to your source checkout. If an action needs to read a source file, declare the file as a dependency and refer to it with a dependency variable such as `%{dep:path/to/file}`. This lets Dune track the dependency and make it available to sandboxed actions.

Dune supports the following variables:

- `project_root` is the root of the current project in the current build context. It is typically the root of your project under `_build/<context>`, and as long as you have a `dune-project` file there, `project_root` is independent of the workspace configuration.
- `workspace_root` is the root of the current workspace in the current build context. For the default build context this is typically `_build/default`. Note that the value of `workspace_root` isn't constant and depends on whether your project is vendored or not.
- `cc` is the C compiler command line (list made of the compiler name followed by its flags) that will be used to compile foreign code. For more details about its content, please see *Flags in Foreign Code*.
- `cxx` is the C++ compiler command line being used in the current build context.
- `ocaml_bin` is the path where `ocamlc` lives.
- `ocaml` is the `ocaml` binary.
- `ocamlc` is the `ocamlc` binary.
- `ocamlopt` is the `ocamlopt` binary.
- `ocaml_version` is the version of the compiler used in the current build context.
- `ocaml_where` is the output of `ocamlc -where`.
- `arch_sixtyfour` is `true` if using a compiler that targets a 64-bit architecture and `false` otherwise.
- `null` is `/dev/null` on Unix or `nul` on Windows.
- `ext_obj`, `ext_asm`, `ext_lib`, `ext_dll`, and `ext_exe` are the file extensions used for various artifacts.
- `ext_plugin` is `.cmxs` if `natdynlink` is supported and `.cma` otherwise.
- `ocaml-config:v` expands to the output of `ocamlc -config-var v` for every configuration variable `v`. Note that Dune processes the output of `ocamlc -config` in order to make it a bit more stable across versions, so the exact set of variables accessible this way might not be exactly the same as what you can see in the output of `ocamlc -config`. In particular, variables added in new OCaml versions need to be registered in Dune before they can be used.
- `profile` is the profile selected via `--profile`.
- `context_name` is the name of the context (`default`, or defined in the workspace file)
- `os_type` is the type of the OS the build is targeting. This is the same as `ocaml-config:os_type`.
- `architecture` is the type of the architecture the build is targeting. This is the same as `ocaml-config:architecture`.
- `model` is the type of the CPU the build is targeting. This is the same as `ocaml-config:model`.
- `system` is the name of the OS the build is targeting. This is the same as `ocaml-config:system`.

The target values of `os_type`, `architecture`, `model`, and `system` come from the OCaml compiler configuration used by the build context. For example, to inspect the values for the current context, run:

```

$ dune exec -- ocamlc -config-var os_type
$ dune exec -- ocamlc -config-var architecture
$ dune exec -- ocamlc -config-var model
$ dune exec -- ocamlc -config-var system

```

On a typical Linux switch these may print `Unix`, `x86_64` or `arm64`, `default`, and `linux` respectively. When writing Dune files, the corresponding `{ocaml-config:...}` variables expand to the same values.

- `ignoring_promoted_rules` is true if `--ignore-promoted-rules` was passed on the command line and false otherwise.
- `dune-warnings` is the list of OCaml warnings that Dune used by default up until version 3.20 of the Dune language when building in the dev profile. This was a larger set of warnings than the default one used by the OCaml compiler, and in version 3.21 of the Dune language the set of warnings used by the dev profile was reverted to the default one used by the compiler. This variable is made available for those users who would like to keep using Dune's stricter warning set. The old behaviour of Dune can be recovered by using the following stanza in a top-level dune file: `(env (dev (flags :standard {dune-warnings})))`.
- `git-sha` expands to the short git SHA of the HEAD commit of the workspace's git repository (equivalent to `git rev-parse --short HEAD`). Expands to the empty string when no commit sha was found. Available since Dune 3.24.
- `<ext>:<path>` where `<ext>` is one of `cmo`, `cmi`, `cma`, `cmx`, or `cmxa`. See *Variables for Artifacts*.
- `env:<var>=<default>` expands to the value of the environment variable `<var>`, or `<default>` if it does not exist. For example, `{env:BIN=/usr/bin}`. Available since Dune 1.4.0.
- There are some Rocq-specific variables detailed in *Rocq-Specific Variables*.

In addition, `(action ...)` fields support the following special variables:

- `target` expands to the one target.
- `targets` expands to the list of target.
- `deps` expands to the list of dependencies.
- `^` expands to the list of dependencies, separated by spaces.
- `dep:<path>` expands to `<path>` (and adds `<path>` as a dependency of the action).
- `exe:<path>` expands to an executable target in the source tree and adds it as a dependency of the action. Use this form to run an in-tree executable by path, for example `{exe:./tool.exe}` or `{exe:./bin/tool.exe}`. The executable does not need a `public_name` and does not need to be installed. This is similar to `dep:<path>`, except that Dune will map the executable to a version that can run on the build machine when cross-compiling.
- `bin:<program>` expands to a runnable path for `program` and adds it as a dependency of the action. If `<program>` is the public name of an executable in the workspace, the expansion is the build-artifact path of the locally built binary; otherwise, `<program>` is looked up in the build context's `PATH` and the expansion is its absolute path. When the resulting relative path is a bare basename (i.e. the binary lives in the action's directory), Dune prepends `./` so that shells like `bash` execute the file directly rather than performing a `PATH` lookup.

When `{bin:<program>}` appears in `(deps ...)`, the action additionally gets an isolated directory prepended to `PATH`, containing a symlink named `<program>` (without the artifact extension) for each declared `{bin:...}` dep. This lets the binary be invoked by its bare name from `(bash ...)` or `(system ...)`.

`{bin:...}` is not required with `(run ...)`: `(run {bin:foo} ...)` and `(run foo ...)` behave the same.

- `bin-available:<program>` expands to true or false, depending on whether `<program>` is available or not.
- `file-available:<path>` expands to true or false, depending on whether the file at `<path>` is available in the current workspace.

- `lib:<public-library-name>:<file>` expands to the file's installation path `<file>` in the library `<public-library-name>`. If `<public-library-name>` is available in the current workspace, the local file will be used, otherwise the one from the *installed world* will be used.
- `lib-private:<library-name>:<file>` expands to the file's build path `<file>` in the library `<library-name>`. Both public and private library names are allowed as long as they refer to libraries within the same project.
- `libexec:<public-library-name>:<file>` is the same as `lib:...` , except when cross-compiling, in which case it will expand to the file from the host build context.
- `libexec-private:<library-name>:<file>` is the same as `lib-private:...`  except when cross-compiling, in which case it will expand to the file from the host build context.
- `lib-available:<library-name>` expands to `true` or `false` depending on whether the library is available or not. A library is available if at least one of the following conditions holds:
  - It's part the *installed world*.
  - It's available locally and is not optional.
  - It's available locally, and all its library dependencies are available.
- `version:<package>` expands to the version of the given package. Packages defined in the current scope have priority over the public packages. Public packages that don't install any libraries will not be detected. How Dune determines the version of a package is described *here*.
- `read:<path>` expands to the contents of the given file.
- `read-lines:<path>` expands to the list of lines in the given file.
- `read-strings:<path>` expands to the list of lines in the given file, unescaped using OCaml lexical convention.
- `ppx:lib1+..+libn` expands to the ppx executable with ppx libraries `lib1` to `libn` linked in. This form also introduces a dependency on this executable.
- `pkg:<package>:<section>:<path>` expands to the path of a file installed by `<package>` in `<section>` at the relative `<path>` within that section. Works with workspace packages, lock-file packages, and installed packages. The supported sections are those listed in *install* (except `misc`).

Added in version 3.24.

The `%{<kind>:...}` forms are what allows you to write custom rules that work transparently, whether things are installed or not.

Note that aliases are ignored by `%{deps}`. The intent of this last form is to reliably read a list of strings generated by an OCaml program via:

```
List.iter (fun s -> print_string (String.escaped s)) l
```

### 4.13.1 Dealing with circular dependencies introduced by variables

If you ever see Dune reporting a dependency cycle that involves a variable such as `%{read:<path>}`, try to move `<path>` to a different directory.

The reason you might see such dependency cycle is because Dune is trying to evaluate the `%{read:<path>}` too early. For instance, let's consider the following example:

```
(rule
  (targets x)
  (enabled_if %{read:y})
  (action ...))
```

(continues on next page)

(continued from previous page)

```
(rule
 (with-stdout-to y (...)))
```

When Dune loads and interprets this file, it decides whether the first rule is enabled by evaluating `%{read:y}`. To evaluate `%{read:y}`, it must build `y`. To build `y`, it must figure out the build rule that produces `y`, and in order to do that, it must first load and evaluate the above dune file. You can see how this creates a cycle.

Some cycles might be more complex. In any case, when you see such an error, the easiest thing to do is move the file that's being read to a different directory, preferably a standalone one. You can use the *subdir* stanza to keep the logic self-contained in the same dune file:

```
(rule
 (targets x)
 (enabled_if %{read:dir-for-y/y})
 (action ...))

(subdir
 dir-for-y
 (rule
 (with-stdout-to y (...))))
```

### 4.13.2 Expansion of Lists

Forms that expand to a list of items, such as `%{cc}`, `%{deps}`, `%{targets}`, or `%{read-lines:...}`, are suitable to be used in `(run <prog> <arguments>)`. For instance in:

```
(run foo %{deps})
```

If there are two dependencies, `a` and `b`, the produced command will be equivalent to the shell command:

```
$ foo "a" "b"
```

If you want both dependencies to be passed as a single argument, you must quote the variable:

```
(run foo "%{deps}")
```

which is equivalent to the following shell command:

```
$ foo "a b"
```

(The items of the list are concatenated with space.) Please note: since `%{deps}` is a list of items, the first one may be used as a program name. For instance:

```
(rule
 (targets result.txt)
 (deps   foo.exe (glob_files *.txt))
 (action (run %{deps})))
```

Here is another example:

```
(rule
 (target foo.exe)
```

(continues on next page)

```
(deps   foo.c)
(action (run %{cc} -o %{target} %{deps} -lfoolib))
```

## 4.14 Dependency Specification

Dependencies in dune files can be specified using one of the following:

- `(:name <dependencies>)` will bind the list of dependencies to the `name` variable. This variable will be available as `%{name}` in actions.
- `(file <filename>)`, or simply `<filename>`, depend on this file.
- `(alias <alias-name>)` depends on the construction of this alias. For instance: `(alias src/runtest)`.
- `(alias_rec <alias-name>)` depends on the construction of this alias recursively in all children directories wherever it is defined. For instance: `(alias_rec src/runtest)` might depend on `(alias src/runtest)`, `(alias src/foo/bar/runtest)`, etc.
- `(glob_files <glob>)` depends on all files matched by `<glob>`. See the *glob* for details.
- `(glob_files_rec <glob>)` is the recursive version of `(glob_files <glob>)`. See the *glob* for details.
- `(source_tree <dir>)` depends on all source files in the subtree with root `<dir>`. A source file is a file that exists in the source tree rather than a target produced by Dune. Source files may still be raw data: for example, files in *data\_only\_dirs* directories are source files, even though Dune does not interpret dune files in those directories. Dune does not traverse directories excluded by the *dirs* stanza. In addition, files and directories whose names begin with `.` are ignored by `source_tree` by default; see *Why does source\_tree ignore files and directories when they begin with a "." (period)?*.
- `(universe)` depends on everything in the universe. This is for cases where dependencies are too hard to specify. Note that Dune will not be able to cache the result of actions that depend on the universe. In any case, this is only for dependencies in the *installed world*. You must still specify all dependencies that come from the workspace.
- `(package <pkg>)` builds the files installed by `<package>` and adds them to the action's environment: bin entries on `PATH`, libraries on `OCAMLPATH`, stublibs on `CAML_LD_LIBRARY_PATH`, and so on. Only the named package is added; transitive package dependencies must be listed separately.
- `(env_var <var>)` depends on the value of the environment variable `<var>`. If this variable becomes set, becomes unset, or changes value, the target will be rebuilt.
- `(sandbox <config>)` requires a particular sandboxing configuration. `<config>` can be one (or many) of:
  - `always`: the action requires a clean environment
  - `none`: the action must run in the build directory
  - `preserve_file_kind`: the action needs the files it reads to look like normal files (so Dune won't use symlinks for sandboxing)
- `(include <file>)` read the s-expression in `<file>` and interpret it as additional dependencies. The s-expression is expected to be a list of the same constructs enumerated here.

In all these cases, the argument supports *Variables*.

### 4.14.1 Named Dependencies

Dune allows a user to organize dependency lists by naming them. The user is allowed to assign a group of dependencies a name that can later be referred to in actions (like the `%{deps}`, `%{target}`, and `%{targets}` built in variables).

One instance where this is useful is for naming globs. Here's an example of an imaginary bundle command:

```
(rule
  (target archive.tar)
  (deps
    index.html
    (:css (glob_files *.css))
    (:js foo.js bar.js)
    (:img (glob_files *.png) (glob_files *.jpg)))
  (action
    (run %{bin:bundle} index.html -css %{css} -js %{js} -img %{img} -o %{target})))
```

Note that a named dependency list can also include unnamed dependencies (like `index.html` in the example above). Also, such user defined names will shadow build in variables, so `(:workspace_root x)` will shadow the built-in `%{workspace_root}` variable.

### 4.14.2 Glob

You can use globs to declare dependencies on a set of files. Note that globs will match files that exist in the source tree as well as buildable targets, so for instance you can depend on `*.cmi`.

Dune supports globbing files in a single directory via `(glob_files ...)` and, starting with Dune 3.0, in all subdirectories recursively via `(glob_files_rec ...)`. The glob is interpreted as follows:

- anything before the last `/` is taken as a literal path
- anything after the last `/`, or everything if the glob contains no `/`, is interpreted using the glob syntax

Absolute paths are permitted in the `(glob_files ...)` term only. It's an error to pass an absolute path (i.e., a path beginning with a `/`) to `(glob_files_rec ...)`.

The glob syntax is interpreted as follows:

- `\<char>` matches exactly `<char>`, even if it's a special character (`*`, `?`, ...).
- `*` matches any sequence of characters, except if it comes first, in which case it matches any character that is not `.` followed by anything.
- `**` matches any character that is not `.` followed by anything, except if it comes first, in which case it matches anything.
- `?` matches any single character.
- `[<set>]` matches any character that is part of `<set>`.
- `[!<set>]` matches any character that is not part of `<set>`.
- `{<glob1>, <glob2>, ..., <globn>}` matches any string that is matched by one of `<glob1>`, `<glob2>`, etc.

Table 5: Glob syntax examples

Syntax	Files matched	Files not matched
<code>x</code>	<code>x</code>	<code>y</code>
<code>\*</code>	<code>*</code>	<code>x</code>
<code>file*.txt</code>	<code>file1.txt, file2.txt</code>	<code>f.txt</code>
<code>*.txt</code>	<code>f.txt</code>	<code>.hidden.txt</code>
<code>a**</code>	<code>aml</code>	<code>a.ml</code>
<code>**</code>	<code>a/b, a.b</code>	(none)
<code>a?.txt</code>	<code>a1.txt, a2.txt</code>	<code>b1.txt, a10.txt</code>
<code>f[xyz].txt</code>	<code>fx.txt, fy.txt, fz.txt</code>	<code>f2.txt, f.txt</code>
<code>f[!xyz].txt</code>	<code>f2.txt, fa.txt</code>	<code>fx.txt, f.txt</code>
<code>a.{ml,mli}</code>	<code>a.ml, a.mli</code>	<code>a.txt, b.ml</code>
<code>../a.{ml,mli}</code>	<code>../a.ml, ../a.mli</code>	<code>a.ml</code>

## 4.15 OCaml Flags

In `library`, `executable`, `executables`, and `env` stanzas, you can specify OCaml compilation flags using the following fields:

- `(flags <flags>)` to specify flags passed to both `ocamlc` and `ocamlopt`
- `(ocamlc_flags <flags>)` to specify flags passed to `ocamlc` only
- `(ocamlopt_flags <flags>)` to specify flags passed to `ocamlopt` only

For all these fields, `<flags>` is specified in the *Ordered Set Language*. These fields all support `(:include ...)` forms.

The value of `:standard` depends on the selected build profile. See *Default OCaml Flags* for the default OCaml flags Dune adds, including the default `-g` in `ocamlc_flags` and `ocamlopt_flags`.

The default value for `(flags ...)` is taken from the environment, as a result it's recommended to write `(flags ...)` fields as follows:

```
(flags (:standard <my options>))
```

## 4.16 Sandboxing

The user actions that run external commands (`run`, `bash`, `system`) are opaque to Dune, so Dune has to rely on manual specification of dependencies and targets. One problem with manual specification is that it's error-prone. It's often hard to know in advance what files the command will read, and knowing a correct set of dependencies is very important for build reproducibility and incremental build correctness.

To help with this problem Dune supports sandboxing. An idealized view of sandboxing is that it runs the action in an environment where it can't access anything except for its declared dependencies.

In practice, we have to make compromises and have some trade-offs between simplicity, information leakage, performance, and portability.

The way sandboxing is currently implemented is that for each sandboxed action we build a separate directory tree (sandbox directory) that mirrors the build directory, filtering it to only contain the files that were declared as dependencies. We run the action in that directory, and then we copy the targets back to the build directory.

You can configure Dune to use sandboxing modes `symlink`, `hardlink`, or `copy`, which determine how the individual files are populated (they will be symlinked, hardlinked, or copied into the sandbox directory).

This approach is very simple and portable, but that comes with certain limitations:

- The actions in the sandbox can use absolute paths to refer to anywhere outside the sandbox. This means that only dependencies on relative paths in the build tree can be enforced/detected by sandboxing.
- The sandboxed actions still run with full permissions of Dune itself, so sandboxing is not a security feature. It won't prevent network access either.
- We don't erase the environment variables of the sandboxed commands. This is something we want to change.
- Performance impact is usually small, but it can get noticeable for fast actions with very large sets of dependencies.

### 4.16.1 Per-Action Sandboxing Configuration

Some actions may rely on sandboxing to work correctly. For example, an action may need the input directory to contain nothing except the input files, or the action might create temporary files that break other build actions.

Some other actions may refuse to work with Sandboxing. For example, if they rely on absolute path to the build directory staying fixed, or if they deliberately use some files without declaring dependencies (this is usually a very bad idea, by the way).

Generally it's better to improve the action so it works with or without sandboxing (especially with), but sometimes you just can't do that.

Things like this can be described using the "sandbox" field in the dependency specification language (see *Dependency Specification*).

### 4.16.2 Global Sandboxing Configuration

Dune always respects per-action sandboxing specification. You can configure it globally to prefer a certain sandboxing mode if the action allows it.

This is controlled by:

- `dune --sandbox <...>` CLI flag (see `man dune-build`)
- `DUNE_SANDBOX` environment (see `man dune-build`)
- `(sandboxing_preference ..)` field in the configuration file (see `man dune-config`)

## 4.17 Locks

Given two rules that are independent, Dune will assume that their associated actions can be run concurrently. Two rules are considered independent if neither of them depend on the other, either directly or through a chain of dependencies. This basic assumption allows Dune to parallelize the build.

However, it is sometimes the case that two independent rules cannot be executed concurrently. For instance, this can happen for more complicated tests. In order to prevent Dune from running the actions at the same time, you can specify that both actions take the same lock:

```
(rule
  (alias runtest)
  (deps  foo)
  (locks m)
  (action (run test.exe %{deps})))

(alias
  (rule  runtest)
  (deps  bar))
```

(continues on next page)

```
(locks m)
(action (run test.exe %{deps})))
```

Dune will make sure that the executions of `test.exe foo` and `test.exe bar` are serialized.

Although they don't live in the filesystem, lock names are interpreted as file names. So for instance, `(with-lock m ...)` in `src/dune` and `(with-lock ../src/m)` in `test/dune` refer to the same lock.

Note also that locks are per build context. So if your workspace has two build contexts setup, the same rule might still be executed concurrently between the two build contexts. If you want a lock that is global to all build contexts, simply use an absolute filename:

```
(rule
  (alias runtest)
  (deps foo)
  (locks /tcp-port/1042)
  (action (run test.exe %{deps})))
```

## 4.18 Diffing and Promotion

You can use Diffing and Promotion flows to compare the outputs of your build in the build directory with the source tree and/or copy the result of the rules into your source tree to store the changes.

## 4.19 Diffing

You can use the `(diff <file1> <file2>)` directive in a rule to compare its output with the version in your source tree. It is useful when your tests produce a file output and you want to make sure that output has not changed.

`(diff <file1> <file2>)` is very similar to `(run diff <file1> <file2>)`. In particular it behaves in the same way:

- When `<file1>` and `<file2>` are equal, it does nothing.
- When they are not, the differences are shown and the action fails.

However, it is different for the following reason:

- The exact command used for diff files can be configured via the `--diff-command` command line argument. Note that it's only called when the files are not byte equals
- By default, it will use `patdiff` if it is installed. `patdiff` is a better diffing program. You can install it via `opam` with:

```
$ opam install patdiff
```

- On Windows, both `(diff a b)` and `(diff? a b)` normalize end-of-line characters before comparing the files.
- Since `(diff a b)` is a built-in action, Dune knows that `a` and `b` are needed, so you don't need to specify them explicitly as dependencies.
- You can use `(diff? a b)` after a command that might or might not produce `b`, for cases where commands optionally produce a *corrected* file
- If `<file1>` doesn't exist, it will compare with the empty file.
- It allows promotion. See below.

Note that `(cmp a b)` does no end-of-line normalization and doesn't print a diff when the files differ. `cmp` is meant to be used with binary files.

## 4.20 Promotion

Promotion relates to copying the output of a Dune rule to your source tree. Common uses include updating rule output after a failed diff (e.g., from a test) or committing output to source control to cut down on dependencies during packaging.

### 4.20.1 Promoting Test or Rule Output After Diffing

Whenever an action `(diff <file1> <file2>)` or `(diff? <file1> <file2>)` fails because the two files are different, Dune allows you to promote `<file2>` as `<file1>` if `<file1>` is a source file and `<file2>` is a generated file.

More precisely, let's consider the following Dune file:

```
(rule
  (with-stdout-to data.out (run ./test.exe)))

(rule
  (alias runtest)
  (action (diff data.expected data.out)))
```

Where `data.expected` is a file committed in the source repository. You can use the following workflow to update your test:

- Update the code of your test.
- Run `dune runtest`. The diff action will fail and a diff will be printed.
- Check the diff to make sure it's what you expect. This diff can be displayed again by running `dune promotion diff`.
- Run `dune promote`. This will copy the generated `data.out` file to `data.expected` directly in the source tree.

You can also use `dune runtest --auto-promote`, which will automatically do the promotion.

### 4.20.2 Automatically Promoting Rule Output Into the Source Tree

Dune rules support a `(mode promote)` directive that will automatically copy their output into your source tree. This approach suits, for example, code documentation generation flows where output needs to be committed to source code control to enable easier browsing, or eliminate dependencies on a code generation step during opam package installation.

More information, including customising when the source is copied, can be found in *rule*.

## 4.21 Package Specification

Installation is the process of copying freshly built libraries, binaries, and other files from the build directory to the system. Dune offers two ways of doing this: via opam or directly via the `install` command. In particular, the installation model implemented by Dune was copied from opam. Opam is the standard OCaml package manager.

In both cases, Dune only know how to install whole packages. A package being a collection of executables, libraries, and other files. In this section, we'll describe how to define a package, how to "attach" various elements to it, and how to proceed with installing it on the system.

### 4.21.1 Declaring a Package

To declare a package, simply add a package stanza to your `dune-project` file:

```
(package
  (name mypackage)
  (synopsis "My first Dune package!")
  (description "\| This is my first attempt at creating
              "\| a project with Dune.
))
```

Once you have done this, Dune will know about the package named `mypackage` and you will be able to attach various elements to it. The package stanza accepts more fields, such as dependencies.

Note that package names are in a global namespace, so the name you choose must be universally unique. In particular, package managers never allow users to release two packages with the same name.

In older projects using Dune, packages were defined by manually writing a file called `<package-name>.opam` at the root of the project. However, it's not recommended to use this method in new projects, as we expect to deprecate it in the future. The right way to define a package is with a package stanza in the `dune-project` file.

See *How to Generate Opam Files from dune-project* for instructions on configuring Dune to automatically generate `.opam` files based on the package stanzas.

### 4.21.2 Attaching Elements to a Package

Attaching an element to a package means declaring to Dune that this element is part of the said package. The method to attach an element to a package depends on the kind of the element. In this subsection, we will go through the various kinds of elements and describe how to attach each of them to a package.

In the rest of this section, `<prefix>` refers to the directory in which the user chooses to install packages. When installing via `opam`, it's `opam` that sets this directory. When calling `dune install`, the installation directory is either guessed or can be manually specified by the user. Defaults directories which replace guessing can be set during the compilation of `dune`.

### 4.21.3 Sites of a Package

When packages need additional resources outside their binary, their location could be hard to find. Moreover, some packages could add resources to another package, e.g., in the case of plugins. These locations are called sites in Dune. One package can define them. During execution, one site corresponds to a list of directories. They are like layers, and the first directories have a higher priority. Examples and precisions are available at *How to Load Additional Files at Runtime*.

#### Libraries

In order to attach a library to a package, merely add a `public_name` field to your library. This is the name that external users of your libraries must use in order to refer to it. Dune requires that a library's public name is either the name of the package it is part of or start with the package name followed by a dot character.

For instance:

```
(library
  (name mylib)
  (public_name mypackage.mylib))
```

After you have added a public name to a library, Dune will know to install it as part of the package it is attached to. Dune installs the library files in a directory `<prefix>/lib/<package-name>`.

If the library name contains dots, the full directory in which the library files are installed is `lib/<comp1>/<comp2>/.../<compn>`, where `<comp1>`, `<comp2>`, ... `<compn>` are the dot-separated component of the public library name. By definition, `<comp1>` is always the package name.

## Executables

Similar to libraries, to attach an executable to a package simply add a `public_name` field to your `executable` stanza or a `public_names` field for `executables` stanzas. Designate this name to match the available executables through the installed PATH (i.e., the name users must type in their shell to execute the program), because Dune cannot guess an executable's relevant package from its public name. It's also necessary to add a `package` field unless the project contains a single package, in which case the executable will be attached to this package.

For instance:

```
(executable
 (name main)
 (public_name myprog)
 (package mypackage))
```

Once `mypackage` is installed on the system, the user will be able to type the following in their shell:

```
$ myprog
```

to execute the program.

## Other Files

For all other kinds of elements, you must attach them manually via an `install` stanza.

## 4.22 Aliases

*Aliases* are build targets that do not correspond to specific files. For example, the `run test` alias corresponds to running tests.

### 4.22.1 Model and Syntax

Dependencies and actions can be attached to an alias. When this alias is requested to be built, these dependencies are built and these actions are executed. Aliases are attached to specific directories.

In commands such as `dune build`, the syntax to refer to the `x` alias is `@x`, for example `dune build @x`. This is why it is common to refer to it as “the `@x` alias”, or “attaching a rule to `@x`”.

Building `@x` will build `x` in all subdirectories of the current directory. This is the most common case, but it is possible to restrict this using different syntaxes:

- `@sub/dir/x` will build `x` in `sub/dir` and its subdirectories.
- `@@x` will build `x` in the current directory only.
- `@@sub/dir/x` will build `x` in `sub/dir` only.

If `dir` is the directory of a *build context*, it restricts the alias to this context.

To summarize, the syntax is:

- `@` (recursive) or `@@` (non-recursive): determine if subdirectories are included
- optional *build context root*: restrict to a particular *build context*
- optional directory: only consider this subdirectory

- alias name

Examples:

- `dune build @_build/foo/runtest` only runs the tests for the `foo` build context
- `dune build @runtest` will run the tests for all build contexts

### 4.22.2 User-Defined Aliases

It is possible to use any name for alias names; it will then be available on the command line. For example, if a Dune file contains the following, then `dune build @deploy` will execute that command.

```
(rule
  (alias deploy)
  (action ./run-deployer.exe))
```

### 4.22.3 Built-In Aliases

Some aliases are defined and managed by Dune itself:

#### @all

This alias corresponds to every known file target in a directory. It is not a hard-coded list of stanza kinds: it includes targets introduced by user rules as well as targets generated for libraries, executables, tests, and other stanzas in that directory.

For example, `@all` can include executable targets such as `foo.exe` or `foo.bc`, library artifacts, and generated files from rule stanzas. Some stanzas may also attach additional dependencies to `@all` explicitly; for example, `melange.emit` attaches its emission targets to `@all`.

The `@default` alias depends recursively on `all` unless it is overridden, so plain `dune build` usually builds `@all` in the workspace subtree.

Since version 2.0 of the dune language, JS targets of executables are no longer included in the `all` alias by default. To get back the old behavior of including the JS targets in `all`, one can add the `js` target to the executable's `modes` field.

#### @default

This alias corresponds to the default argument for `dune build`: `dune build` is equivalent to `dune build @@default` (`@@` indicates a *non-recursive alias*). Similarly, `dune build dir` is equivalent to `dune build @@dir/default`.

When a directory doesn't explicitly define what the `default` alias means via an *alias* stanza, the following implicit definition is assumed:

```
(alias
  (name default)
  (deps (alias_rec all)))
```

But if such a stanza is present in the dune file in a directory, it will be used instead. For example, if the following is present in `tests/dune`, `dune build tests` will run tests there:

```
(alias
  (name default)
  (deps (alias_rec runtest)))
```

## Interaction With Tests

The default alias is distinct from the `@runtest` alias. Plain `dune build` builds `@@default`. It doesn't run actions attached only to `runtest` unless they are requested explicitly, for example with `dune runtest` or `dune build @runtest`.

However, the implicit definition of `default` depends recursively on `@all`, and `all` includes file targets produced by tests, executables, libraries, and rules. As a result, plain `dune build` can still build test executables or generated test artifacts in subdirectories.

To keep a tests directory out of plain `dune build`, override `default` in a parent directory so that it only depends on the subtrees that should be part of the ordinary build. For example, a project with `bin/`, `lib/`, and `tests/` can put this in the top-level dune file:

```
(alias
 (name default)
 (deps
  (alias_rec bin/all)
  (alias_rec lib/all)))
```

Overriding `default` inside `tests/dune` would only affect commands such as `dune build tests` or `dune build @@tests/default`; it would not change the implicit recursive dependency used by the parent directory's `default` alias.

## @revdep, @revdep-check, @revdep-runtest, @revdep-install

`@revdep` builds the default alias of every local target that transitively depends on any library defined in the current directory. The reverse dependencies can be other libraries, executables, or tests; Dune walks the dependency graph and schedules the appropriate alias in each dependent directory.

Variants map to other standard aliases in the dependents:

- `@revdep-check` builds the `@check` alias of every reverse dependency.
- `@revdep-runtest` builds the `@runtest` alias of every reverse dependency.
- `@revdep-install` builds the `@install` alias of every reverse dependency.

## Usage examples

- `dune build @src/revdep` builds the default alias of all targets that depend (directly or transitively) on any library declared in `src`.
- `dune build @src/revdep-check` builds the corresponding `@check` alias for those reverse dependencies.

## Notes

- These aliases only do work when explicitly requested; they do not add overhead to regular builds.
- If multiple libraries live in the same dune file, their reverse dependencies are combined, so `@dir/revdep` covers dependents of any library in `dir`.

## @opam

Building this alias generates `*.opam` files for projects that use `generate_opam_files`.

This is the dedicated alias for opam file generation. The `install` and `runtest` aliases depend on it, so building either of those will also refresh the generated `*.opam` files.

With (lang dune 3.23) or newer, when a checked-in \*.opam file differs from the generated one, Dune records the change as a promotion. Apply it with `dune promote`.

With older Dune language versions, building `@opam` promotes the generated \*.opam files to the source tree automatically.

### @install

Building this alias will create the \*.install files used by the *opam integration*. In turn, these depend on installable files.

### @pkg-install

This alias is only relevant when using Dune with *package management* (see *OCaml Package Management With Dune*). Running `dune build @pkg-install` will fetch the dependencies described in the `depends` field of your `dune-project` (see *package*) and build them. It will not build your project.

Indeed, if you need to build the project, you need to use the regular `dune build` command. Note that if the dependencies have not already been fetched and downloaded, `dune build` will **also** take care of getting and building them.

#### Note

`dune build @pkg-install` is particularly useful when you are building projects using per-layer caching systems, e.g., Docker images. Using this alias, you will be able to cache the dependencies building stage as they change less regularly.

If you are building the `@pkg-install` alias in a repository where package management is not activated, the command will fail.

#### See also

*How Package Management Works*

### @empty

The *empty* alias contains no targets.

As of Dune language version 3.20, user-defined *rule* and *alias* stanzas are no longer permitted to extend the *empty* alias.

### @check

This alias corresponds to the set of targets necessary for development tools to work correctly. For example, it will build \*.cmi, \*.cmt, and \*.cmti files so that Merlin and `ocaml-lsp-server` can be used in the project. It is also useful in the development loop because it will catch compilation errors without executing expensive operations such as linking executables.

Starting from (lang dune 3.23), this alias also generates and promotes a `compile_commands.json` file to the workspace root for projects that contain C or C++ foreign stubs. This file is used by `clangd` and `ccls` for code navigation and diagnostics. See *Editor Support for C/C++ Stubs* for details.

#### See also

`@ocaml-index` for a fast feedback loop that also indexes the project.

## @ocaml-index

This alias corresponds to the set of targets necessary for development tools to provide project-wide queries such as “get all references of this value”. These targets are indexes built using the required *ocaml-index* binary. Since this alias also includes the *\*.cmi*, *\*.cmt*, and *\*.cmti* files usually built by `check`, it can be used in most projects as a replacement to get a fast feedback loop while maintaining the indexes up-to-date.

## @runtest

Actions that run tests are attached to this alias. For example this convention is used by the `(test)` stanza.

Building `@runtest` directly uses normal *alias* target semantics. In a workspace with multiple contexts, an unqualified alias target is built in every configured context.

The `dune runtest` command, also available as `dune test`, has different command-line behavior: it accepts test path arguments and selects a build context for source paths. See *Running Tests* for details.

### See also

*Writing and Running Tests*

## @fmt

This alias is used by formatting rules: when it is built, code formatters will be executed (using *promotion*).

`dune fmt` is a shortcut for `dune build @fmt --auto-promote`.

It is possible to build on top of this convention. If some actions are manually attached to the `fmt` alias, they will be executed by `dune fmt`.

Example:

```
(rule
 (with-stdout-to
  data.json.formatted
  (run jq . ${dep:data.json})))

(rule
 (alias fmt)
 (action
  (diff data.json data.json.formatted)))
```

## @lint

This alias runs the linters configured by *lint field*.

For example, `dune build @lint` runs linters in the current directory and its subdirectories. If a PPX linter reports corrections, Dune displays them as diffs that can be applied with `dune promote`.

## @unused-libs

This alias is used to detect unused entries in the `libraries` field of executables and stanzas.

## @doc

This alias builds HTML documentation for public libraries and package `.mld` pages. It does not document executable stanzas as API pages. Use `@doc-private` to build documentation for private libraries.

### ➔ See also

*Generating Documentation*

## @doc-private

This alias builds documentation for all libraries, both public & private.

## @doc-json

This alias builds documentation for public libraries as JSON files. These are produced by `odoc`'s option `--as-json` and can be consumed by external tools.

The generated files mirror the HTML documentation hierarchy and use the `.html.json` suffix. For example, the package index is generated at `_build/default/_doc/_html/index.html.json`.

### ➔ See also

*Generating Documentation*

## 4.23 Foreign Stubs

You can specify foreign sources using the `foreign_stubs` field of the `library` and `executable` stanzas. For example:

```
(library
 (name lib)
 (foreign_stubs (language c) (names src1 src2))
 (foreign_stubs (language cxx) (names src3) (flags -O2)))
```

Here we declare an OCaml library `lib`, which contains two C sources `src1` and `src2`, and one C++ source, `src3`, which need to be compiled with `-O2`. These source files will be compiled and packaged with the library, along with the link-time flags to be used when linking the final executables. When matching names to source files, Dune treats `*.c` files as C sources, and `*.cpp`, `*.cc`, and `*.cxx` files as C++ sources.

Here is a complete list of supported subfields:

- `language` specifies the source language, where `c` means C and `cxx` means C++. In the future, more languages may be supported.
- `names` specifies the *names* of source files. When specifying a source file, omit the extension and any relative parts of the path; Dune will scan all library directories to find all matching files and raise an error if multiple source files map to the same object name. If you need to have multiple object files with the same name, you can package them into different *Foreign Archives* via the `foreign_archives` field. This field uses the *Ordered Set Language* where the `:standard` value corresponds to the set of names of all source files whose extensions match the specified language.
- `flags` are passed when compiling source files. This field is specified using the *Ordered Set Language*, where the `:standard` value comes from the environment settings `c_flags` and `cxx_flags`, respectively. Note that, for C stubs, Dune unconditionally adds the flags present in the OCaml config fields `ocamlc_flags` and

`ocamlc_cppflags` to the compiler command line. This behavior can be disabled since Dune 2.8 via the `dune-project` option `use_standard_c_and_cxx_flags`.

- `include_dirs` are tracked as dependencies and passed to the compiler via the `-I` flag. You can use *Variables* in this field and refer to a library source directory using the `(lib library-name)` syntax. Additionally, the syntax `(include filename)` can be used to specify a file containing additional arguments to `(include_dirs ...)`. The named file can either contain a single path to be added to this list of include directories, or an S-expression listing additional `(include_dirs ...)` arguments (the `(lib ...)` and `(include ...)` syntax is also supported in files included in this way). For example, `(include_dirs dir1 (lib lib1) (lib lib2) (include inc1) dir2)` specifies the directory `dir1`, the source directories of `lib1`, and `lib2`, the list of directories contained in the file `inc1`, and the directory `dir2`, in this order. Some examples of possible contents of the file `inc1` are:
  - `dir3` which would add `dir3` to the list of include directories
  - `((lib lib3) dir4 (include inc2))` which would add the source directory of the library `lib3`, the directory `dir4`, and the result of recursively including the contents of the file `inc2`. The contents of included directories are tracked recursively, e.g., if you use `(include_dirs dir)` and have headers `dir/base.h` and `dir/lib/lib.h`, they both will be tracked as dependencies.
- **`extra_deps` specifies any other dependencies that should be tracked. This** is useful when dealing with `#include` statements that escape into a parent directory like `#include "../a.h"`.

### 4.23.1 Mode-Dependent Stubs

Since Dune 3.5, it is possible to use different foreign stubs when building in *native* or *byte* mode. This feature needs to be activated by adding `(using mode_specific_stubs 0.1)` in the `dune-project` file.

Then it is allowed to use the `mode` field when describing `foreign_stubs`. If the same stub is defined twice, Dune will automatically chose the correct one. This allows the use of different sets of flags or even different source files from which the stubs are built.

```
(executable
 (name main)
 (modes native byte_complete)
 (foreign_stubs
  (language c)
  (mode byte)
  (names c_stubs))
 (foreign_stubs
  (language c)
  (mode native)
  (flags :standard -DNATIVE_CODE) ; A flag specific to native builds
  (names c_stubs))) ; This could be the name of an implementation
                    ; specific to native builds
```

Note that, as of version 0.1 of this extension, this mechanism does not work for `foreign_archives`.

## 4.24 Foreign Archives

Foreign archives are a way to link archives that are separately compiled.

They are particularly useful when embedding a library written in a foreign language and/or built with another build system. See *Foreign Build Sandboxing* for more details.

To use this feature, use the `foreign_archives` field of the corresponding `library` or `executable` stanza. For example:

```
(library
 (name lib)
 (foreign_stubs (language c) (names src1 src2))
 (foreign_stubs (language cxx) (names src3) (flags -O2))
 (foreign_archives arch1 some/dir/arch2))
```

Here, in addition to *Foreign Stubs*, we also specify foreign archives `arch1` and `arch2`, where the latter is stored in a subdirectory `some/dir`.

You can build a foreign archive manually, e.g., using a custom rule as described in *Foreign Build Sandboxing*, or ask Dune to build it via the `foreign_library` stanza:

```
(foreign_library
 (archive_name arch1)
 (language c)
 (enabled_if true)
 (names src4 src5)
 (extra_objects obj1)
 (include_dirs headers))
```

This asks Dune to compile C source files `src4` and `src5` with headers tracked in the `headers` directory and put the resulting object files into an archive `arch1`, whose full name is typically `libarch1.a` for static linking and `dllarch1.so` for dynamic linking.

By default, Dune builds and installs both the static (`lib*.a`) and dynamic (`dll*.so`) versions of a foreign archive. To statically link all foreign archives into the runtime system instead (so no `dll*.so` is produced), set `(disable_dynamically_linked_foreign_archives true)` in your *context*.

The `foreign_library` stanza supports all *Foreign Stubs* fields. The `archive_name` field specifies the archive's name. You can refer to the same archive name from multiple OCaml libraries and executables, so a foreign archive is a bit like a foreign library, hence the name of the stanza. The `enabled_if` field has the same meaning as in the *library* stanza. The `extra_objects` field specifies additional object files to be included. Dune will look for `obj1.o` in this case. `extra_objects` uses the *Ordered Set Language* and supports `(:include ...)` forms.

## 4.25 Extra Objects

It's possible to specify native object files to be packaged with OCaml libraries or linked into OCaml executables. Do this by using the `extra_objects` field of the `library` or `executable` stanzas. For example:

```
(executable
 (public_name main)
 (extra_objects foo bar))

(rule
 (targets foo.o bar.o)
 (deps foo.c bar.c)
 (action (run ocamlcopt %{deps})))
```

This example builds an executable which is linked against a pair of native object files, `foo.o` and `bar.o`. The `extra_objects` field takes a list of object names, which correspond to the object file names with their path and extension omitted (in contrast, `(rule)` manipulates file names, so the extension needs to be provided).

In this example, the sources corresponding to the objects (`foo.c` and `bar.c`) are assumed to be present in the same directory as the OCaml source code, and a custom rule is used to compile the C source code into object files using `ocamlc_opt`. This is not necessary; one can instead compile foreign object files manually and place them next to the OCaml source code.

## 4.26 Flags in Foreign Code

Depending on the `use_standard_c_and_cxx_flags` option, the base `:standard` set of flags for C will contain only `ocamlc_cflags` or both `ocamlc_cflags` and `ocamlc_cppflags`.

There are multiple levels where one can declare custom flags (using the *Ordered Set Language*), and each level inherits the flags of the previous one in its `:standard` set:

- In the global `env` definition of a `dune-workspace` file
- In the `per-context env` definitions in a `dune-workspace` file
- In the `env` definition of a `dune` file
- In a `foreign_stubs`, `foreign_library` or `extra_objects` field of an executable or a library.

The `%{cc}` *variable* will contain the flags from the first three levels only.

For example, to add a flag to all C stubs in the `dev` profile, write:

```
(env
 (dev
  (c_flags (:standard -DMY_DEBUG_STUBS))))
```

To add flags only to one `foreign_stubs` field, write:

```
(library
 (name mylib)
 (foreign_stubs
  (language c)
  (names mystubs)
  (flags (:standard -DMYLIB_STUBS))))
```

If the flags come from a generated file, for example from a `dune-configurator` script that queried `pkg-config`, include them with `(:include ...)`:

```
(library
 (name mylib)
 (foreign_stubs
  (language c)
  (names mystubs)
  (flags (:standard (:include c_flags.sexp))))
 (c_library_flags (:standard (:include c_library_flags.sexp))))
```

## 4.27 Command Line Interface

This is a short overview of the commands available in Dune. Reference documentation for each command is available through `dune COMMAND --help`.

### **dune build**

Build the given targets, or the default ones.

### **dune cache**

Manage the shared cache of build artifacts.

#### **dune cache size**

Query the size of the Dune cache.

#### **dune cache trim**

Trim the Dune cache.

### **dune clean**

Clean the project.

### **dune rocq**

Command group related to Rocq.

#### **dune rocq top**

Execute a Rocq toplevel with the local configuration.

### **dune describe**

Describe the workspace.

#### **dune describe aliases**

Print aliases in a given directory. Works similarly to ls.

#### **dune describe env**

Print the environment of a directory.

#### **dune describe external-lib-deps**

Print out the external libraries needed to build the project. It's an approximated set of libraries.

#### **dune describe installed-libraries**

Print out the libraries installed on the system.

#### **dune describe opam-files**

Print information about the opam files that have been discovered.

#### **dune describe package-entries**

prints information about the entries per package.

#### **dune describe pp**

Build a given file and print the preprocessed output.

#### **dune describe rules**

Dump rules.

#### **dune describe targets**

Print targets in a given directory. Works similarly to ls. The directory may be a path in the source tree, or a build-only directory under `_build/` (such as `_build/default/.lib.objs`).

#### **dune describe workspace**

Print a description of the workspace's structure. If some directories are provided, then only those directories of the workspace are considered.

### **dune diagnostics**

Fetch and return errors from the current build.

**dune exec**

Execute a command in a similar environment as if installation was performed.

Dune's options and the executed program's options share the same command line. If an argument to the program starts with `-`, separate it from Dune's options with `--`:

```
$ dune exec ./tool.exe -- --program-option
```

Equivalently, put the separator before the whole command:

```
$ dune exec -- ./tool.exe --program-option
```

**dune fmt**

Format source code.

**dune format-dune-file**

Format dune files.

**dune help**

Additional Dune help.

**dune init**

Command group for initializing Dune components.

**dune init executable**

Initialize a binary executable.

**dune init library**

Initialize an OCaml library.

**dune init project**

Initialize a whole OCaml project.

**dune init test**

Initialize a test harness.

**dune install**

Install packages defined in workspace.

**dune installed-libraries**

Print out libraries installed on the system.

**dune ocaml**

Command group related to OCaml.

**dune ocaml dump-dot-merlin**

Print Merlin configuration.

**dune ocaml merlin**

Command group related to Merlin.

**dune ocaml merlin dump-config**

Prints the entire content of the Merlin configuration for the given folder in a user friendly form.

**dune ocaml merlin start-session**

Start a Merlin configuration server.

**dune ocaml ocaml-merlin**

Start a Merlin configuration server.

**dune ocaml top**

Print a list of toplevel directives for including directories and loading `.cma` files.

**dune ocaml top-module**

Print a list of toplevel directives for loading a module into the toplevel.

**dune ocaml utop**

Load library in UTop.

**dune ocaml-merlin**

Start a Merlin configuration server.

**dune printenv**

Print the environment of a directory.

**dune promotion**

Control how changes are propagated back to source code.

**dune promotion apply**

Promote files from the last run.

**dune promotion diff**

List promotions to be applied.

**dune promote**

A command alias for `dune promotion apply`.

**dune rpc**

Dune's RPC mechanism. Experimental.

**dune rules**

Dump rules.

**dune runtest**

Run tests.

**dune test**

A command alias for `dune runtest`.

**dune shutdown**

Cancel and shutdown any builds in the current workspace.

**dune subst**

Substitute watermarks in source files.

**dune tools**

Added in version 3.17.

Command group for managing developer tools. See *dune tools* for details.

** Warning**

The `dune tools` command group is **experimental**. Its subcommands, flags, and behavior may change in future versions of Dune without notice.

**dune tools exec**

Run a developer tool, automatically building it if needed.

**dune tools install**

Install a developer tool without running it.

**dune tools which**

Print the path to a developer tool's executable.

**dune tools env**

Print a command to add developer tool directories to your PATH.

**dune top**

Print a list of toplevel directives for including directories and loading `.cma` files.

**dune uninstall**

Uninstall packages defined in the workspace.

**dune upgrade**

Upgrade projects across major Dune versions.

**dune utop**

Load library in UTop.

## 4.28 dune tools

**Warning**

The `dune tools` command group is **experimental**. Its subcommands, flags, and behavior may change in future versions of Dune without notice. The internal APIs that power these commands are also subject to change.

If you encounter issues, please [open an issue](#) in the Dune issue tracker.

`dune tools` is a command group for managing developer tools. Developer tools are programs that are useful for working on a project's source code but are not required for building or deploying the project itself. Examples include `ocamlformat` for formatting your source code, `ocaml-lsp-server` which provides a Language Server Protocol implementation for OCaml, and `utop` an improved toplevel (REPL).

Dune can automatically resolve, lock, build, and run these tools using package management. Each tool is managed in its own lock directory separate from the project.

### 4.28.1 Supported Tools

The following developer tools are currently supported:

Tool	Executable	Package
ocamlformat	ocamlformat	ocamlformat
odoc	odoc	odoc
ocaml-lsp-server	ocaml lsp	ocaml-lsp-server
utop	utop	utop
ocamlearlybird	ocamlearlybird	earlybird
odig	odig	odig
opam-publish	opam-publish	opam-publish
dune-release	dune-release	dune-release
ocaml-index	ocaml-index	ocaml-index
merlin	ocamlmerlin	merlin

### **Note**

`dune tools exec` supports all tools above, but `odoc` and `utop` also have their own dedicated subcommands commands, which run the tools with additional configuration:

- `dune utop [DIR]` discovers libraries in the given directory (defaults to working directory), configures library search paths, and generates a custom `findlib.conf` so that `utop` can find dependencies within `_build`. `utop` needs to be installed via `opam` or `dune tools install utop` before invoking this command.
- `dune ocaml doc` builds documentation for the current project using the `@doc` alias. As with `utop`, `odoc` must be installed before use.

## 4.28.2 Subcommands

### `dune tools exec`

```
$ dune tools exec <tool> [-- <args>...]
```

Run a developer tool. Dune will automatically resolve, lock, and build the tool if it has not been installed yet, then execute it. This is intended to be called by text editors and other tooling that needs to invoke a dev tool.

All positional arguments are forwarded to the tool's executable. .. code-block:: console

```
$ dune tools exec ocamlformat $ dune tools exec -help
```

### `dune tools install`

```
$ dune tools install <tool>
```

Install a developer tool without running it. Dune will resolve dependencies, create a separate lock directory, and build the tool. This is also useful for pre-installing tools so that later `exec` or `which` calls are fast.

```
$ dune tools install ocamlformat
$ dune tools install ocaml lsp
```

### `dune tools which`

```
$ dune tools which <tool> [--allow-not-installed]
```

Print the path to a developer tool's executable. By default, this errors if the tool has not been installed yet.

Flags:

- **--allow-not-installed**  
Print the path where the tool would be installed, even if it does not exist yet.

```
$ dune tools which ocamlformat
_build/.dev-tools.locks/ocamlformat/ocamlformat/target/bin/ocamlformat

$ dune tools which merlin --allow-not-installed
_build/.dev-tools.locks/merlin/merlin/target/bin/ocamlmerlin
```

### dune tools env

```
$ dune tools env [--fish]
```

Print a shell command that, when evaluated, adds all dev tool binary directories to your PATH. This allows running dev tools directly as commands without going through `dune tools exec`.

Flags:

- **--fish**  
Emit a `fish_add_path` command suitable for the fish shell instead of the default POSIX `export PATH=.` form.

For POSIX-compatible shells (bash, zsh, etc.):

```
$ eval $(dune tools env)
```

For the fish shell:

```
$ eval (dune tools env --fish)
```

#### **Note**

The design of `dune tools` is still being fleshed out and may be reworked. See [dune#13457](#) for ongoing discussion.

## 4.28.3 See Also

- *How to Customize Lock Directories of Developer Tools* for customizing the lock directory configuration of developer tools.

## 4.29 Dune Libraries

### 4.29.1 Configurator

Configurator is a small library designed to query features available on the system in order to generate configuration for Dune builds. Such generated configuration is usually in the form of command line flags, generated headers, and stubs, but there are no limitations on this.

Configurator allows you to query for the following features:

- Variables defined in `ocamlc -config`,
- `pkg-config` flags for packages,
- Test features by compiling C code,

- Extract compile time information such as `#define` variables.

Configurator is designed to be cross-compilation friendly and avoids *running* any compiled code to extract any of the information above.

Configurator started as an [independent library](#), but now lives in dune. It is released as the package `dune-configurator`.

### Usage

We'll describe configurator with a simple example. Everything else can be easily learned by studying [configurator's API](#).

To use Configurator, write an executable that will query the system using Configurator's API and output a set of targets reflecting the results. For example:

```
module C = Configurator.V1

let clock_gettime_code = [|
#include <time.h>

int main(void)
{
    struct timespec ts;
    clock_gettime(CLOCK_REALTIME, &ts);
    return 0;
}
|]

let () =
  C.main ~name:"foo" (fun c ->
    let has_clock_gettime =
      C.c_test c clock_gettime_code ~link_flags:["-lrt"] in

    C.C_define.gen_header_file c ~fname:"config.h"
    [ "HAS_CLOCK_GETTIME", Switch has_clock_gettime ]);
```

Usually, the module above would be named `discover.ml`. The next step is to invoke it as an executable and tell Dune about the targets that it produces:

```
(executable
 (name discover)
 (libraries dune-configurator))

(rule
 (targets config.h)
 (action (run ./discover.exe)))
```

Another common pattern is to produce a flags file with Configurator and then use this flag file using `:include`:

```
(library
 (name mylib)
 (foreign_stubs (language c) (names foo))
 (c_library_flags (:include (flags.sexp))))
```

For this, generate the list of flags for your library (for example, using `Configurator.V1.Pkg_config`), and then write them to a file: in the above example, `flags.sexp` with `Configurator.V1.write_flags "flags.sexp" flags`.

## Upgrading From the Old Configurator

The old Configurator is the independent `Configurator` opam package. It's now deprecated, and users are encouraged to migrate to Dune's own Configurator. The advantage of the transition include:

- No extra dependencies,
- No need to manually pass `-ocamlc` flag,
- New Configurator is cross-compilation compatible.

The following steps must be taken to transition from the old Configurator:

- Mentions of the `configurator` opam package should be replaced with `dune-configurator`.
- The library name `configurator` should be changed `dune-configurator`.
- The `-ocamlc` flag in rules that runs Configurator scripts should be removed. This information is now passed automatically by Dune.
- The new Configurator API is versioned explicitly. The version that's compatible with old Configurator is under the `V1` module. Hence, to transition one's code, it's enough to add this module alias:

```
module Configurator = Configurator.V1
```

### 4.29.2 `dune-build-info` Library

Dune can embed build information such as versions in executables via the special `dune-build-info` library. This library exposes some information about how the executable was built, such as the version of the project containing the executable or the list of statically linked libraries with their versions. Printing the version at which the current executable was built is as simple as:

```
Printf.printf "version: %s\n"
  (match Build_info.V1.version () with
  | None -> "n/a"
  | Some v -> Build_info.V1.Version.to_string v)
```

You can specify the project version using the `version` field in the `dune-project` file. For example:

```
(version 1.2.3)
```

For more details, refer to *version*.

For libraries and executables from development repositories that don't have version information written directly in the `dune-project` file, the version is obtained by querying the version control system. For instance, the following Git command is used in Git repositories:

```
$ git describe --always --dirty --abbrev=7
```

which produces a human readable version string of the form `<version>-<commits-since-version>-<hash>[-dirty]`.

Note that in the case where the version string is obtained from the version control system, the version string will only be written in the binary once it's installed or promoted to the source tree. In particular, if you evaluate this expression as part of your package build, it will return `None`. This ensures that committing doesn't hurt your development experience. Indeed, if Dune stored the version directly inside the freshly built binaries, then every time you commit your code, the version would change and Dune would need to rebuild all the binaries and everything that depends on them, such as tests. Instead, Dune leaves a placeholder inside the binary and fills it during installation or promotion.

### 4.29.3 (Experimental) Dune Action Plugin

*This library is experimental and no backwards compatibility is implied. Use at your own risk.*

`dune-action-plugin` provides a monadic interface to express program dependencies directly inside the source code. Programs using this feature should be declared using `dynamic-run` instead of usual `run`.

## 4.30 Dune Caches

Dune implements several different caches:

- The build cache contains build artifacts that are shared across different workspaces. This is the main cache, interacted with via the `dune cache` command.
- The toolchains cache contains the compiler (and compiler variants) when they can't be installed like regular packages, due to them not being relocatable.
- The revision store is a git repository containing all revisions fetched via git.

Their location is as follows:

Name	Default status	Default location on Unix	Alternate location (has priority)
Build cache	Enabled	<code>XDG_CACHE_HOME/dune/db</code>	<code>DUNE_CACHE_ROOT/db</code>
Toolchains cache	Enabled	<code>XDG_CACHE_HOME/dune/toolchains</code>	<code>DUNE_CACHE_ROOT/toolchains</code>
Revision store	Enabled	<code>XDG_CACHE_HOME/dune/git-repo</code>	<code>DUNE_CACHE_ROOT/git-repo</code>

#### Note

On Windows, you can replace `XDG_CACHE_HOME` by `%LOCALAPPDATA%\Microsoft\Windows\Temporary Internet Files`

## 4.31 How Caching Works

Before executing a build rule, Dune hashes the rule and looks up that hash in the shared cache, and if it finds a matching entry, Dune skips the rule's execution and restores the results in the current build directory. This can greatly speed up builds when different workspaces share code, as well as when switching branches or simply undoing some changes within the same workspace.

## 4.32 Configuration

There are three ways to configure the Dune cache. Choose the one that is more convenient for you:

- Add `(cache <setting>)` to your Dune configuration file (`~/.config/dune/config` by default).
- Set the environment variable `DUNE_CACHE` to `<setting>`
- Run Dune with the `--cache=<setting>` flag.

Here, `<setting>` must be one of:

- `disabled`: disables the Dune cache completely.

- `enabled-except-user-rules`: enables the Dune cache, but excludes user-written rules. This setting is a conservative choice that can avoid breaking rules whose dependencies are not correctly specified. Currently the default.
- `enabled`: enables the Dune cache unconditionally.

By default, Dune stores the cache in your `XDG_CACHE_HOME` directory on \*nix systems and `%LOCALAPPDATA%\Microsoft\Windows\Temporary Internet Files\dune` on Windows. You can change the default location by setting the environment variable `DUNE_CACHE_ROOT`.

## 4.33 Cache Storage Mode

Dune supports two modes of storing and restoring cache entries: *hardlink* and *copy*. If your file system supports hard links, we recommend that you use the *hardlink* mode, which is generally more efficient and reliable.

### 4.33.1 The *hardlink* Mode

By default, Dune uses hard links when storing and restoring cache entries. This is fast and has zero disk space overhead for files that still live in a build directory. There are two disadvantages of this mode:

- The cache storage must be on the same partition as the build tree.
- A cache entry can be corrupted by modifying the hard link that points to it from the build directory. To reduce the risk of cache corruption, Dune systematically removes write permissions from all build results. It is worth noting that modifying files in the build directory is a bad practice anyway.
- The cache does not preserve exact Unix permission bits. Cache entries are identified by file contents and Dune's executable flag, not by the complete file mode. When two artifacts differ only in ignored permission bits, Dune may deduplicate them into the same cache entry. In *hardlink* mode, all copies then share one inode and therefore one set of permissions.

### 4.33.2 The *copy* Mode

If you specify (`cache-storage-mode copy`) in the configuration file, Dune will copy files to and from the cache instead of using hard links. This mode is slower and has higher disk space usage. On the positive side, it is more portable and doesn't have the disadvantages of the *hardlink* mode (see above).

You can also set or override the storage mode via the environment variable `DUNE_CACHE_STORAGE_MODE` and the command line flag `--cache-storage-mode`.

## 4.34 Trimming the Cache

Storing all historically produced build results in the cache is infeasible, so you'll need to occasionally trim the cache. To do that, run the `dune cache trim --size=BYTES` command. This will remove the oldest used cache entries to keep the cache overhead below the specified size. By "overhead" we mean the cache entries whose hard link count is equal to 1, i.e., which aren't used in any build directory. Trimming cache entries whose hard link count is greater than 1 would not free any disk space.

Note that previous versions of Dune, cache provided a "cache daemon" that could periodically trim the cache. The current version doesn't require an additional daemon process, so this automated trimming functionality is no longer provided.

## 4.35 Reproducibility

### 4.35.1 Reproducibility Check

While the main purpose of Dune cache is to speed up build times, it can also be used to check build reproducibility. By specifying (`cache-check-probability` FLOAT) in the configuration file, or running Dune with the `--cache-check-probability=FLOAT` flag, you instruct Dune to re-execute randomly chosen build rules and compare their results with those stored in the cache. If the results differ, the rule is not reproducible, and Dune will print out a corresponding warning.

### 4.35.2 Non-Reproducible Rules

Some build rules are inherently not reproducible because they involve running non-deterministic commands that, for example, depend on the current time or download files from the Internet. To prevent Dune from caching such rules, mark them as non-reproducible by using (`deps (universe)`). Please see *Dependency Specification* for more details.

## 4.36 The Rocq Prover Build Language

### 4.36.1 Introduction

Dune can build Rocq theories and plugins with additional support for extraction and `.mlg` file preprocessing.

A *Rocq theory* is a collection of `.v` files that define Rocq modules whose names share a common prefix. The module names reflect the directory hierarchy.

Rocq theories may be defined using *rocq.theory* stanzas, or be auto-detected by Dune by inspecting Rocq’s install directories.

A *Rocq plugin* is an OCaml *library* that Rocq can load dynamically at runtime. Plugins are typically linked with the Rocq OCaml API.

Rocq plugins need to be “public” libraries in Dune’s terminology, that is to say, they must declare a `public_name` field; this is necessary as they are located using `ocamlfind`.

A *Rocq project* is an informal term for a *dune-project* containing a collection of Rocq theories and plugins.

The `.v` files of a theory need not be present as source files. They may also be Dune targets of other rules.

To enable Rocq support in a Dune project, specify the *Rocq language version* in the *dune-project* file. For example, adding

```
(using rocq 0.14)
```

to a *dune-project* file enables using the `rocq.theory` stanza and other `rocq.*` stanzas. See the *Dune Rocq language* section for more details.

### 4.36.2 rocq.theory

The Rocq theory stanza is very similar in form to the OCaml *library* stanza:

```
(rocq.theory
 (name <module_prefix>)
 (package <package>)
 (synopsis <text>)
 (modules <ordered_set_lang>)
 (plugins <ocaml_plugins>)
 (flags <rocq_flags>)
```

(continues on next page)

(continued from previous page)

```
(modules_flags <flags_map>)
(generate_project_file)
(rocqdep_flags <rocqdep_flags>)
(rocqdoc_flags <rocqdoc_flags>)
(rocqdoc_header <rocqdoc_header>)
(rocqdoc_footer <rocqdoc_footer>)
(no_corelib)
(mode <rocq_build_mode>)
(theories <rocq_theories>))
```

The stanza builds all the `.v` files in the given directory and its subdirectories if the `include-subdirs` stanza is present.

When a `.v` file has a corresponding `.expected` file, Rocq's standard output is captured instead of being output to the terminal, and the output is diffed with the `.expected` file as part of the `runtest` alias. When the output differs, it can be promoted as with, e.g., `cram` tests. (Appeared in *Rocq lang 0.12*)

For usage of this stanza, see the *Examples of Rocq Projects*.

The semantics of the fields are:

- `<module_prefix>` is a dot-separated list of valid Rocq module names and determines the module scope under which the theory is compiled (this corresponds to Rocq's `-R` option).

For example, if `<module_prefix>` is `foo.Bar`, the theory modules are named `foo.Bar.module1`, `foo.Bar.module2`, etc. Note that modules in the same theory don't see the `foo.Bar` prefix in the same way that OCaml wrapped libraries do.

For compatibility, *Rocq lang 1.0* installs a theory named `foo.Bar` under `foo/Bar`. Also note that Rocq supports composing a module path from different theories, thus you can name a theory `foo.Bar` and a second one `foo.Baz`, and Dune composes these properly. See an example of *a multi-theory Rocq project* for this.

- The `modules` field allows one to constrain the set of modules included in the theory, similar to its OCaml counterpart. Modules are specified in Rocq notation. That is to say, `A/b.v` is written `A.b` in this field.
- If the `package` field is present, Dune generates install rules for the `.vo` files of the theory. `pkg_name` must be a valid package name.

Note that *Rocq lang 1.0* uses the Rocq install setup, where all packages share a common root namespace and install directory, `lib/rocq/user-contrib/<module_prefix>`, as is customary in the Make-based Rocq package ecosystem.

For compatibility, Dune also installs, under the `user-contrib` prefix, the `.cmxs` files that appear in `<ocaml_plugins>`. This will be dropped in future versions.

- `<rocq_flags>` are passed to `rocq c` as command-line options. `:standard` is taken from the value set in the `(rocq (flags <flags>))` field in `env` profile. See *env* for more information.
- `<flags_map>` is a list of pairs of valid Rocq module names and a list of `<rocq_flags>`. Note that if a module is present here, the `:standard` variable will be bound to the value of `<rocq_flags>` effective for the theory. This way it is possible to override the default flags for particular files of the theory, for example:

```
(rocq.theory
 (name Foo)
 (modules_flags
  (bar (:standard \ -quiet))))
```

It is more common to just use this field to *add* some particular flags, but that should be done using `(:standard <flag1> <flag2> ...)` as to propagate the default flags.

- `<rocqdep_flags>` are extra user-configurable flags passed to `rocq dep`. The default value for `:standard` is empty. This field exists for transient use-cases, in particular disabling `rocq dep` warnings, but it should not be used in normal operations.
- `<rocqdoc_flags>` are extra user-configurable flags passed to `rocq doc`. The default value for `:standard` is `--toc`. The `--html` or `--latex` flags are passed separately depending on which mode is target. See the section on *Rocq Documentation* for more information.
- `<rocqdoc_header>` is a file passed to `rocq doc` using the `--with-header` option, to configure a custom HTML header for the generated HTML pages.
- `<rocqdoc_footer>` is a file passed to `rocq doc` using the `--with-footer` option, to configure a custom HTML footer for the generated HTML pages.
- `(no_corelib)` disables Dune's automatic Corelib handling. When this field is present, Corelib is not added as an implicit dependency and `Corelib.Init.Prelude` is not imported by default. See the section on *The Rocq Standard Library* for details.

Added in version 3.24.

#### Note

In Rocq language 0.13 and earlier, this setting was written as `(stdlib no)`. In Rocq language 0.14 and later, `stdlib` was deleted: use `(no_corelib)` instead of `(stdlib no)`, and remove `(stdlib yes)` entirely.

- If the `plugins` field is present, Dune will pass the corresponding flags to Rocq so that `rocq dep` and `rocq c` can find the corresponding OCaml libraries declared in `<ocaml_plugins>`. This allows a Rocq theory to depend on OCaml plugins. The field must contain a public library name.
- Your Rocq theory can depend on other theories — globally installed or defined in the current workspace — by adding the theories names to the `<rocq_theories>` field. Then, Dune will ensure that the depended theories are present and correctly registered with Rocq.

See *Locating Theories* for more information on how Rocq theories are located by Dune.

- The `<rocq_build_mode>` field does control what Rocq objects are built. Allowed values are `vo` or `vos`.

By default, Dune will build Rocq's `.vo` and `.glob` files.

If Rocq was configured with `-native-compiler yes`, Dune will also build the corresponding `cmxs` native files.

You may disable the compilation of native objects in this case by specifying `(mode vo)`, which can bring an important speedup in some cases.

If `(mode vos)` is set, Dune will instead output `.vos` interface files, which skip proof checking.

- If the `(generate_project_file)` is present, a `_RocqProject` file is generated in the Rocq theory's directory (it is promoted to the source tree). This file should be suitable for editor compatibility, and it provides an alternative to using `dune rocq top`. These files are attached to the `@rocqproject` alias, so `dune build @rocqproject` updates and promotes all of them. This feature is however limited in two ways: it is incompatible with the `(modules_flags ...)` field, and it cannot be used for two Rocq theories declared in the same directory.

## Rocq Dependencies

When a Rocq file `a.v` depends on another file `b.v`, Dune is able to build them in the correct order, even if they are in separate theories. Under the hood, Dune uses `rocq dep` to resolve these dependencies, which is why it is called once per theory.

## Rocq Documentation

Given a *rocq.theory* stanza with name `A`, Dune will produce two *directory targets*, `A.html/` and `A.tex/`. HTML or LaTeX documentation for a Rocq theory may then be built by running `dune build A.html` or `dune build A.tex`, respectively (if the *dune file* for the theory is the current directory).

There are also two aliases `@doc` and `@doc-latex` that will respectively build the HTML or LaTeX documentation when called. These will determine whether or not Dune passes a `--html` or `--latex` flag to `rocq doc`.

Further flags can also be configured using the `(rocqdoc_flags)` field in the `rocq.theory` stanza. These will be passed to `rocq doc` and the default value is `:standard` which is `--toc`. Extra flags can therefore be passed by writing `(rocqdoc_flags :standard --body-only)` for example.

When building the HTML documentation, flags `(rocqdoc_header)` and `(rocqdoc_footer)` can also be used to configure a custom HTML header or footer respectively.

## Recursive Qualification of Modules

If you add:

```
(include_subdirs qualified)
```

to a *dune* file, Dune considers all the modules in the directory and its subdirectories, adding a prefix to the module name in the usual Rocq style for subdirectories. For example, file `A/b/C.v` becomes the module `A.b.C`.

## How Dune Locates and Builds theories

Dune organises its knowledge about Rocq theories in 3 databases:

- Scope database: A Dune *scope* is a part of the project sharing a single common `dune-project` file. In a single scope, any theory in the database can depend on any other theory in that database as long as their visibilities are compatible. A public theory for example cannot depend on a private theory.
- Public theory database: The set of all scopes that Dune knows about is termed a *workspace*. Only public theories coming from scopes are added to the database of all public theories in the current workspace.

The public theory database allows theories to depend on theories that are in a different scope. Thus, you can depend on theories belonging to another *dune-project* as long as they share a common scope under another *dune-project* file or a *dune-workspace* file.

Doing so is usually as simple as placing a Rocq project within the scope of another. This process is termed *composition*. See the *interproject composition* example.

Inter-project composition allows Dune to compute module dependencies using a fine granularity. In practice, this means that Dune will only build the parts of a depended theory that are needed by your project.

- Installed theory database: If a theory cannot be found in the list of workspace-public theories, Dune will try to locate the theory in the list of installed locations Rocq knows about.

This list is built using the output of `rocq c --config`. That command reports the Rocq library root path under the key `ROCQLIB` (note: this key is still named `COQLIB` in current Rocq releases and will eventually be renamed). Each path under `ROCQLIB/user-contrib` is scanned to build the database of installed theories.

If the `ROCQPATH` environment variable is set, its paths are also scanned for installed theories. This is useful for theories installed outside the standard `user-contrib` location.

Note that, for backwards compatibility purposes, installed theories do not have to be installed or built using Dune. Dune tries to infer the name of the theory from the installed layout. This is ambiguous in the sense that a file-system layout of `a/b` will provide theory names `a` and `a.b`.

Resolving this ambiguity in a backwards-compatible way is not possible, but future versions of Dune Rocq support will provide a way to improve this.

Rocq's core library gets a special status in Dune. The location at `ROCQLIB/theories` will be assigned an entry with the theory name `Corelib`, and added to the dependency list implicitly. This can be disabled with the `(no_corelib)` field in the `rocq.theory` stanza. Even if disabled, the `Corelib` theory is still available as an explicit dependency that can be added to the `theories` field.

Note that `Stdlib` is a *separate* installed theory and is never added implicitly. To use `Stdlib` modules, add `(theories Stdlib)` to your stanza regardless of the `(no_corelib)` setting.

The databases above are used to locate a theory dependencies. Note that Dune has a complete global view of every file involved in the compilation of your theory and will therefore rebuild if any changes are detected.

### Public and Private Theories

A *public theory* is a `rocq.theory` stanza that is visible outside the scope of a *dune-project* file.

A *private theory* is a `rocq.theory` stanza that is limited to the scope of the *dune-project* file it is in.

A private theory may depend on both private and public theories; however, a public theory may only depend on other public theories.

By default, all `rocq.theory` stanzas are considered private by Dune. In order to make a private theory into a public theory, the `(package )` field must be specified.

```
(rocq.theory
 (name private_theory))

(rocq.theory
 (name private_theory)
 (package rocq-public-theory))
```

### Limitations

- `.v` files always depend on the native OCaml version of the Rocq binary and its plugins, unless the natively compiled versions are missing.
- A `foo.mlpack` file must be present in directories of locally defined plugins for things to work. `rocq dep`, which is used internally by Dune, will recognize a plugin by looking at the existence of an `.mlpack` file, as it cannot access (for now) Dune's library database. This is a limitation of `rocq dep`. See the [example plugin](#) or the [this template](#).

This limitation will be lifted soon, as newer versions of `rocq dep` can use `findlib`'s database to check the existence of OCaml libraries.

### Rocq Language Version

The Rocq lang can be modified by adding the following to a *dune-project* file:

```
(using rocq 0.14)
```

The supported Rocq language versions (not the version of Rocq) are:

- **0.11:** Support for the Rocq Prover; most important changes are:
  - all deprecated features in `(lang coq 0.10)` have been removed.
  - Dune won't install `.cmxs` files in `user-contrib` (along `.vo` files) anymore.
  - `(mode native)` is not allowed anymore. It is the default if Rocq was configured with `native compute enabled`.
  - `COQPATH` is not recognized anymore, use `ROCQPATH`.

- 0.12: Support for output tests.
- 0.13: `rocq.extraction` now uses `extracted_files` instead of `extracted_modules`, supporting extraction to languages other than OCaml.
- 0.14: `(stdlib ...)` was replaced by `(no_corelib)`.

### Rocq Language Version 1.0

Guarantees with respect to stability are not yet provided, but we intend that the (0.14) version of the language becomes 1.0. The 1.0 version of Rocq lang will commit to a stable set of functionality. All the features below are expected to reach 1.0 unchanged or minimally modified.

### The Rocq Standard Library

Rocq's standard library is divided into two separately-installed theories:

- **Corelib:** The core prelude, providing fundamental data types and basic logic. Corresponds to the path `ROCQLIB/theories` in the Rocq installation. See [The Rocq Reference Manual](#) for more information on the contents of this library.

By default, Dune adds `Corelib` as an implicit dependency and automatically imports `Corelib.Init.Prelude`. This makes basic syntax and types available without any explicit `Require`. Add `(no_corelib)` to disable both the implicit dependency and the default prelude import.

In Rocq language 0.13 and earlier, this setting was written as `(stdlib no)`. When upgrading to Rocq language 0.14, replace `(stdlib no)` with `(no_corelib)` and remove `(stdlib yes)`.

```
; Rocq language 0.13 and earlier
(rocq.theory
 (name MyTheory)
 (stdlib no))

; Rocq language 0.14 and later
(rocq.theory
 (name MyTheory)
 (no_corelib))
```

- **Stdlib:** The standard library, providing additional modules such as `Strings`, `ZArith`, `Lists`, etc. This is a regular installed theory, typically named `Stdlib` in the `user-contrib` directory. See [Rocq Stdlib](#) for more information.

#### Warning

`Stdlib` is *never* added implicitly regardless of the `(no_corelib)` setting. Use `(theories Stdlib)` to add `Stdlib`.

### 4.36.3 rocq.extraction

Rocq may be instructed to *extract* sources as part of the compilation process by using the `rocq.extraction` stanza:

```
(rocq.extraction
 (prelude <name>)
 (extracted_files <filenames>)
 <optional-fields>)
```

- `(prelude <name>)` refers to the Rocq source that contains the extraction commands.

- `(extracted_files <filenames>)` is the list of files Dune expects the extraction to produce. Each entry is a filename with its extension, for example OCaml (`.ml`, `.mli`), Haskell (`.hs`), or Scheme (`.scm`).

Added in version 3.23.

### **Note**

`(extracted_modules <names>)` is the predecessor to `extracted_files`, available in (rocq 0.12) and below. It accepts bare module names and expects Dune to produce both a `.ml` and `.mli` file for each. Users upgrading from 0.12 should replace `(extracted_modules M1 ... Mn)` with `(extracted_files M1.ml M1.mli ... Mn.ml Mn.mli)`.

- `<optional-fields>` are flags, `no_corelib`, `theories`, and `plugins`. All of these fields have the same meaning as in the `rocq.theory` stanza.

The extracted sources can then be used in `executable` or `library` stanzas as any other sources.

Note that the sources are extracted to the directory where the `prelude` file lives. Thus the common placement for the OCaml stanzas is in the same `dune` file.

**Warning:** using Rocq's `Cd` command to work around problems with the output directory is not allowed when using extraction from Dune. Moreover the `Cd` command has been deprecated in Coq 8.12.

### 4.36.4 rocq.pp

Authors of Rocq plugins often need to write `.mlg` files to extend the Rocq grammar. Such files are preprocessed using the `rocq pp-mlg` subcommand. To help plugin authors avoid writing boilerplate, we provide a `(rocq.pp ...)` stanza:

```
(rocq.pp
 (modules <ordered_set_lang>))
```

This will run `rocq pp-mlg` on all the `.mlg` files in `<ordered_set_lang>`.

### 4.36.5 Examples of Rocq Projects

Here we list some examples of some basic Rocq project setups in order.

#### Simple Project

Let us start with a simple project. First, make sure we have a `dune-project` file with a `Rocq lang` stanza present:

```
(lang dune 3.25)
(using rocq 0.14)
```

Next we need a `dune` file with a `rocq.theory` stanza:

```
(rocq.theory
 (name myTheory))
```

Finally, we need a Rocq `.v` file which we name `A.v`:

```
(** This is my def *)
Definition mydef := nat.
```

Now we run `dune build`. After this is complete, we get the following files:

```

.
├── A.v
├── _build
│   ├── default
│   │   ├── A.glob
│   │   ├── A.v
│   │   └── A.vo
│   └── log
├── dune
└── dune-project

```

## Multi-Theory Project

Here is an example of a more complicated setup:

```

.
├── A
│   ├── AA
│   │   └── aa.v
│   ├── AB
│   │   └── ab.v
│   └── dune
├── B
│   ├── b.v
│   └── dune
└── dune-project

```

Here are the *dune* files:

```

; A/dune
(include_subdirs qualified)
(rocq.theory
 (name A))

; B/dune
(rocq.theory
 (name B)
 (theories A))

```

Notice the `theories` field in B allows one *rocq.theory* to depend on another. Another thing to note is the inclusion of the *include\_subdirs* stanza. This allows our theory to have *multiple subdirectories*.

Here are the contents of the *.v* files:

```

(* A/AA/aa.v is empty *)

(* A/AB/ab.v *)
Require Import AA.aa.

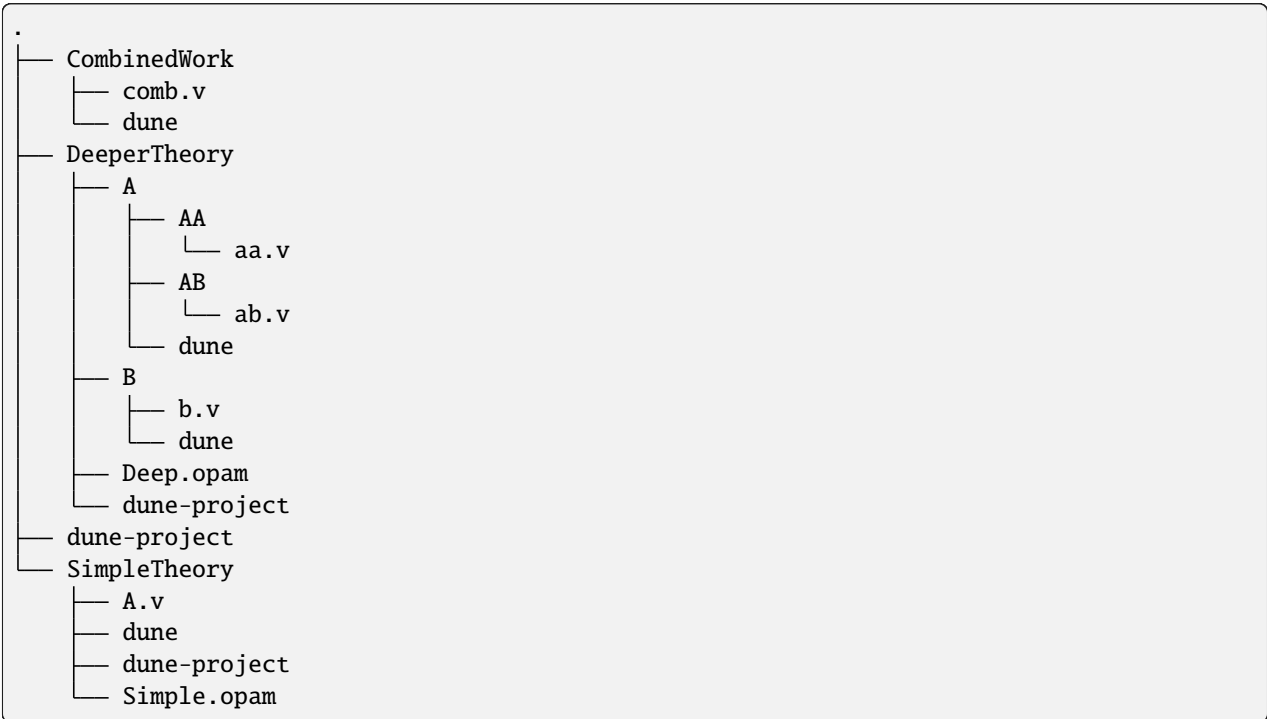
(* B/b.v *)
From A Require Import AB.ab.

```

This causes a dependency chain `b.v -> ab.v -> aa.v`. Now we might be interested in building theory B, so all we have to do is run `dune build B`. Dune will automatically build the theory A since it is a dependency.

## Composing Projects

To demonstrate the composition of Rocq projects, we can take our previous two examples and put them in project which has a theory that depends on theories in both projects.



The file `comb.v` looks like:

```

(* Files from DeeperTheory *)
From A.AA Require Import aa.
(* In Rocq, partial prefixes for theory names are enough *)
From A Require Import ab.
From B Require Import b.

(* Files from SimpleTheory *)
From myTheory Require Import A.
    
```

We are referencing Rocq modules from all three of our previously defined theories.

Our `dune` file in `CombinedWork` looks like:

```

(rocq.theory
 (name Combined)
 (theories myTheory A B))
    
```

As you can see, there are dependencies on all the theories we mentioned.

All three of the theories we defined before were *private theories*. In order to depend on them, we needed to make them *public theories*. See the section on *Public and Private Theories*.

## Composing With Installed Theories

We can also compose with theories that are installed. If we wanted to have a theory that depends on the Rocq theory `mathcomp.ssreflect` we can add the following to our stanza:

```
(rocq.theory
 (name my_mathcomp_theory)
 (theories mathcomp.ssreflect))
```

Note that `mathcomp` on its own would also work, since there would be a `matchcomp` directory in `user-contrib`, however it would not compose locally with a `rocq.theory` stanza with the `mathcomp.ssreflect` name (in case one exists). So it is advisable to use the actual theory name. Dune is not able to validate theory names that have been installed since they do not include their Dune metadata.

## Building Documentation

Following from our last example, we might wish to build the HTML documentation for A. We simply do `dune build A/A.html/`. This will produce the following files:

```
A
├── AA
│   ├── aa.glob
│   ├── aa.v
│   └── aa.vo
├── AB
│   ├── ab.glob
│   ├── ab.v
│   └── ab.vo
├── A.html
│   ├── A.AA.aa.html
│   ├── A.AB.ab.html
│   ├── rocqdoc.css
│   ├── index.html
│   └── toc.html
```

We may also want to build the LaTeX documentation of the theory B. For this we can call `dune build B/B.tex/`. If we want to build all the HTML documentation targets, we can use the `@doc` alias as in `dune build @doc`. If we want to build all the LaTeX documentation then we use the `@doc-latex` alias instead.

## Rocq Plugin Project

Let us build a simple Rocq plugin to demonstrate how Dune can handle this setup.

```
.
├── dune-project
├── src
│   ├── dune
│   ├── hello_world.ml
│   ├── my_plugin.mlpack
│   └── syntax.mlg
├── theories
│   ├── dune
│   └── UsingMyPlugin.v
```

Our `dune-project` will need to have a package for the plugin to sit in, otherwise Rocq will not be able to find it.

```
(lang dune 3.25)
(using rocq 0.14)

(package
  (name my-rocq-plugin)
  (synopsis "My Rocq Plugin")
  (depends rocq-core))
```

Now we have two directories, `src/` and `theories/` each with their own *dune* file. Let us begin with the plugin *dune* file:

```
(library
  (name my_plugin)
  (public_name my-rocq-plugin.plugin)
  (synopsis "My Rocq Plugin")
  (flags :standard -rectypes -w -27)
  (libraries rocq-core.vernac))

(rocq.pp
  (modules syntax))
```

Here we define a library using the *library* stanza. Importantly, we declared which external libraries we rely on and gave the library a `public_name`, which Rocq will use to identify the plugin.

The *rocq.pp* stanza allows `src/syntax.mlg` to be preprocessed, which for reference looks like:

```
DECLARE PLUGIN "my-rocq-plugin.plugin"

VERNAC COMMAND EXTEND Hello CLASSIFIED AS QUERY
| [ "Hello" ] -> { Feedback.msg_notice Pp.(str Hello_world.hello_world) }
END
```

Together with `hello_world.ml`:

```
let hello_world = "hello world!"
```

They make up the plugin. There is one more important ingredient here and that is the `my_plugin.mlpack` file, needed to signal rocq dep the existence of `my_plugin` in this directory. An empty file suffices. See [this note on .mlpack files](#).

The file for `theories/` is a standard *rocq.theory* stanza with an included `libraries` field allowing Dune to see `my-rocq-plugin.plugin` as a dependency.

```
(rocq.theory
  (name MyPlugin)
  (package my-rocq-plugin)
  (plugins my-rocq-plugin.plugin))
```

Finally, our `.v` file will look something like this:

```
Declare ML Module "my-rocq-plugin.plugin".

Hello.
```

Running `dune build` will build everything correctly.

### 4.36.6 Running a Rocq Toplevel

Dune supports running a Rocq toplevel binary such as `rocq top`, which is typically used by editors such as RocqIDE or Proof General to interact with Rocq.

The following command:

```
$ dune rocq top <file> -- <args>
```

runs a Rocq toplevel (`rocq top` by default) on the given Rocq file `<file>`, after having recompiled its dependencies as necessary. The given arguments `<args>` are forwarded to the invoked command. For example, this can be used to pass a `-emacs` flag to `rocq top`.

A different toplevel can be chosen with `dune rocq top --toplevel CMD <file>`. Note that using `--toplevel echo` is one way to observe what options are actually passed to the toplevel. These options are computed based on the options that would be passed to the Rocq compiler if it was invoked on the Rocq file `<file>`.

In certain situations, it is desirable to not rebuild dependencies for `.v` files but still pass the correct flags to the toplevel. For this reason, a `--no-build` flag can be passed to `dune rocq top` which will skip any building of dependencies.

#### Limitations

- Only files that are part of a stanza can be loaded in a Rocq toplevel.
- When a file is created, it must be written to the file system before the Rocq toplevel is started.
- When new dependencies are added to a file (via a Rocq `Require` vernacular command), it is in principle required to save the file and restart the Rocq toplevel process.

### 4.36.7 Rocq-Specific Variables

There are some special variables that can be used to access data about the Rocq configuration. These are:

- `#{rocq:version}` the version of Rocq.
- `#{rocq:version.major}` the major version of Rocq (e.g., `9.1.2` gives `9`).
- `#{rocq:version.minor}` the minor version of Rocq (e.g., `9.1.2` gives `1`).
- `#{rocq:version.suffix}` the suffix version of Rocq (e.g., `9.1.2` gives `.2` and `9.2+rc1` gives `+rc1`).
- `#{rocq:ocaml-version}` the version of OCaml used to compile Rocq.
- `#{rocq:rocqlib}` the output of `ROCQLIB` from `rocq c --config`.
- `#{rocq:rocq_native_compiler_default}` the output of `ROCQ_NATIVE_COMPILER_DEFAULT` from `rocq c --config`.

See *Variables* for more information on variables supported by Dune.

### 4.36.8 Rocq Environment Fields

The `env` stanza has a `(rocq <rocq_fields>)` field with the following values for `<rocq_fields>`:

- `(flags <flags>)`: The default flags passed to `rocq c`. The default value is `-q`. Values set here become the `:standard` value in the `(rocq.theory (flags <flags>))` field.
- `(rocqdep_flags <flags>)`: The default flags passed to `rocq dep`. The default value is empty. Values set here become the `:standard` value in the `(rocq.theory (rocqdep_flags <flags>))` field. As noted in the documentation of the `(rocq.theory (rocqdep_flags <flags>))` field, changing the `rocq dep` flags is discouraged.

- (`rocqdoc_flags <flags>`): The default flags passed to `rocq doc`. The default value is `--toc`. Values set here become the `:standard` value in the `(rocq.theory (rocqdoc_flags <flags>))` field.
- (`rocqdoc_header <file>`): The default HTML header passed to `rocq doc` via the `--with-header` flag. Values set here become the `:standard` value in the `(rocq.theory (rocqdoc_header <file>))` field.
- (`rocqdoc_footer <file>`): The default HTML footer passed to `rocq doc` via the `--with-footer` flag. Values set here become the `:standard` value in the `(rocq.theory (rocqdoc_footer <file>))` field.

### 4.37 Dune RPC

Starting from dune 3.0, dune's watch mode also runs an RPC server. This is a general mechanism introduced to integrate with various dune functionality. Some use cases we have in mind are:

- Text editors that would like to receive dune's error reporting
- In the future, custom actions that need to communicate with dune

There's no fixed scope for RPC, and we encourage users to submit requests to cover more functionality.

The purpose of this documentation is to explain how RPC works, and how to connect to it as a client. More concrete information such as what requests are available is in [Dune\\_rpc](#)

#### 4.37.1 *dune-rpc* library

We provide a client library `dune-rpc` to make it easy to write clients. The library has a versioned interface and we guarantee to maintain its stability. The library documentation describes the guarantees in more detail.

RPC clients written with this library are guaranteed to work for all versions of the RPC server greater than it. In other words, a client using version `X` will work with all servers `>= X`. However, it will not work with any servers `< X`.

The library contains a client implementation parameterised over a concurrency monad. The `dune-rpc-lwt` package provides an instantiation of the RPC client defined in `dune-rpc` using `lwt` to implement the concurrency monad. It is of course possible to instantiate the dune RPC client using alternative concurrency implementations.

The `dune-rpc` library provides an API to do the following:

- Initialise an RPC session over a channel
- Send RPC requests and notifications
- Handle notifications received from the server
- Definitions of available requests, notifications, and their associated types.
- Subscribe to streams of build progress events

[Here](#) is an example of a simple Dune RPC client that uses the `dune-rpc` and `dune-rpc-lwt` packages.

#### 4.37.2 Connecting

To connect to Dune's RPC server, it needs to be started in watch mode. There are two ways of doing this:

- Run `dune build --passive-watch-mode` to start an RPC server which will listen for requests without starting a build by itself...
- Or run `dune build --watch` to start a build server that rebuilds the project's default target when source files change and *also* starts an RPC server that listens for requests.

Then `dune rpc build .` will connect to it, trigger a build, and report status.

## 4.38 Packages

*Packages* are the units that are made installable. Executables and libraries that have a `(public_name)` are installable, and need to be attached to a package.

### 4.38.1 How Dune Knows About Packages

Packages can be defined in two ways:

- If a `<package-name>.opam` file is found, it signals the existence of the `<package-name>` package to Dune. The contents of the opam file are not interpreted by Dune, unless *package management* is *enabled*.
- In `dune-project`, a *(package)* stanza with `(name <package-name>)` defines the `<package-name>` package.

#### Note

If you want to define public elements but are not interested in publishing to opam, it is not necessary to write empty `*.opam` files. Instead, prefer declaring it in `dune-project` as a *(package)* with just a `(name)`.

### 4.38.2 Generating Opam Files

In the latter case, if *(generate\_opam\_files)* is present in `(dune-project)`, Dune will generate `<package-name>.opam` using the metadata in found in *(package)* stanzas and global fields such as `(source)`.

#### See also

*How to Generate Opam Files from dune-project*

If a `<package-name>.opam.template` file is found, the fields found in this opam file are added to the generated file, overriding the generated ones in case of collision. This can be used as an escape hatch to add arbitrary opam fields.

#### See also

*package* for fields supported by `dune-project`

### 4.38.3 What Dune Installs

Dune will install the following files:

- `(library)` stanzas which have a `(public_name)`. The package name is the first component of the dot-separated public name: `(public_name <package-name>)` and `(public_name <package-name>.sub)` will look for a package named `<package-name>` to install their contents.
- `(executable)` and `(test)` stanzas with a `(public_name)`. The package to install the files in is determined by the `(package)` field, which can be omitted if there is only one package.
- `(install)` stanzas provide a low-level mechanism to install arbitrary files.
- Following the conventions of `odig`, files named `README*`, `CHANGE*`, `HISTORY*` and `LICENSE*` are installed in the packages defined by opam files in the same directory. These files can be generated by user rules, So for instance, a changelog generated by a user rule will be automatically installed as well.

## 4.39 Findlib Integration

Dune integrates with `findlib` so that it is possible to use a dependency built with Dune in a project that does not use Dune, or vice versa.

### ➔ See also

*The OCaml Ecosystem* explains the role of `findlib` and its relation with Dune.

To do so, Dune both interprets and generates `META` files.

### 4.39.1 How Dune Interprets `META` files

`META` files use the concept of *predicates*, which can be used to change the interpretation of the directives the files contain. However, Dune does not expose this to the user.

Instead, Dune interprets `META` files assuming the following set of predicates:

- `mt`: refers to a library that can be used with or without threads. Dune will force the threaded version.
- `mt_posix`: forces the use of POSIX threads rather than VM threads. VM threads are deprecated and will soon be obsolete.
- `ppx_driver`: when a library acts differently depending on whether it's linked as part of a driver or meant to add a `-ppx` argument to the compiler, choose the former behavior.

### 4.39.2 The Special Case of the OCaml Compiler

Libraries installed by the compiler are a special case: when the OCaml compiler is older than version 5.0, it does not include `META` files. In that situation, Dune uses its own internal database.

### 4.39.3 How Dune Generates `META` Files

When Dune builds a library, it generates a corresponding `META` file automatically. Usually, there is nothing to do. However, for the rare cases where a specific `META` file is needed, or to ease the transition of a project to Dune, it is possible to write/generate a specific one.

If a `META.<package>.template` is found in the same directory as `<package>.OPAM`, the generated `META.<package>` file will be produced by starting from the contents of `META.<package>.template` and replacing lines of the form `# DUNE_GEN` with the contents of the `META` it would normally generate.

For instance, to add `field = "..."` to the generated `META` file of package `pkg`, you can create a file named `META.pkg.template` with the following contents:

```
# DUNE_GEN
blah = "..."
```

## EXPLANATION

These documents explain how certain feature works, or how Dune integrates with the rest of the OCaml ecosystem.

### 5.1 The Dune Mental Model

It is not strictly necessary to understand Dune’s underlying model to use it; but knowing how it works under the hood will help writing build rules, and also help understand some errors and what’s possible with Dune.

**Note**

This document is a simplification of the reality: the actual rules might be different, it does not touch rule loading and glosses over how caching works, but should be a useful tool to build an understanding of Dune.

#### 5.1.1 How Dune Works

The building block of Dune is the *rule*:

A *rule* reads *dependencies* and writes *targets* using an *action* (and it can be attached to *aliases*).

When `dune build` is executed, it will first read the project’s `dune` files to determine the rules that apply to the project. Once it has done this, it will determine what actions it needs to execute to build the required targets.

#### 5.1.2 An Example

Let’s take the following example.

- there’s a CLI tool written in OCaml.
- it has some build-time configuration stored in `config.json`.
- it has an integration test, in which the tool is executed with `testdata.txt` as input.

#### Configuration Generation

To express the generation of the configuration module we could write:

```
(rule
  (deps convert/json2ml.exe config.json)
  (target config.ml)
  (action
    (run convert/json2ml.exe config.json -o config.ml)))
```

This rule will:

- read its dependencies: `convert/json2ml.exe` and `config.json`
- and write its target: `config.ml`
- using an action: `(run convert/json2ml.exe config.json -o config.ml)`

This rule is very explicit: we write a stanza for a single Dune rule.

### Building the Executable

In contrast, to describe the compilation of the executable, we would write:

```
(executable
 (name tool)
 (modules main config))
```

Here, we use Dune’s abstractions. Dune knows about the OCaml compilation model: the modules need to be compiled and linked together. So it will generate the following rules under the hood:

- one rule to compile the `Main` module:
  - it will read its dependency: `main.ml`
  - and write its output: `main.cmx`
  - using an action: `(run ocamlc -c main.ml)`
- one rule to compile the `Config` module:
  - it will read its dependency: `config.ml`
  - and write its output: `config.cmx`
  - using an action: `(run ocamlc -c config.ml)`
- one rule to link the `tool.exe` executable:
  - it will read its dependencies: `main.cmx` and `config.cmx`
  - and write its output: `tool.exe`
  - using an action: `(run ocamlc -o tool.exe main.cmx config.cmx)`

Note that in this example, some files are targets of a rule and dependencies of another (`.cmx` files). We are unlikely to ever interact with them directly, so it can also be useful to think of the `(executable)` stanza as a group of rules with `main.ml` and `config.ml` as inputs and `tool.exe` as output.

### Running the Tests

Some rules do not produce any output file, but we’re still interested in running their actions. A test is a good example: we want the build process to exit with an error code if the action fails. In that case, the rule does not have targets, but we “attach” it to an *alias*, `runtest` in this case. This gives us a way of requesting this rule to be executed. As we are about to see, rules are executed lazily by asking for their targets to be built, so we would not be able to execute such rules.

```
(rule
 (deps tool.exe testdata.txt)
 (alias runtest)
 (action
  (run tool.exe testdata.txt)))
```

This rule:

- reads its dependencies: `tool.exe` and `testdata.txt`
- writes no targets
- using an action: `(run tool.exe testdata.txt)`
- (and it is attached to `runtest`)

### 5.1.3 What to Build

Dune can build *files* and *aliases*. These can be found on the command line:

- `dune build tool.exe` will build the `tool.exe` file.
- `dune build @example` will build the `example` alias.
- `dune build tool.exe @example` will build both the file `tool.exe` and the `example` alias.
- `dune runtest` will build the `runtest` alias using the test command's path handling. Passing a directory will build all tests in that directory. Passing the path to a cram test will run that test individually. See [Running Tests](#) for the differences from `dune build @runtest`.
- `dune build` is a shortcut for `dune build @@default`: it will build the default alias in the current directory (by default the `all` alias).

In other words, each `dune build` or `dune runtest` command always corresponds to a list of files and aliases to build.

#### See also

*Reference information on aliases*

### 5.1.4 How Dune Interprets Rules

We have now seen that Dune sets up rules for a project, and that every build command has a list of files and aliases that we are asking to build.

Now let's see how this request is processed:

- to build a file, Dune will first check if it is in the source tree. In that case, there is nothing to do. Otherwise, it will check if it is the target of a rule. In that case, it will execute this rule. (Dune will raise an error in other cases: if the file is both in the source tree and the target of a rule, or if it is neither)
- to build an alias, Dune will execute all the rules that are attached to this alias.
- to execute a rule, Dune will first build all the dependencies (files or aliases) of this rule. Then it will execute the action attached to the rule. When Dune is about to execute an action, it checks (in various caches) if it executed it before on the same set of dependencies, and, if yes, it can skip executing it and reuse the previous result.

In the case of our example, if we call `dune runtest`, Dune will consider all rules attached to the `runtest` alias. In this case it is just the integration test rule. It needs to build its dependencies, `tool.exe` and `testdata.txt`. The latter is present in the source tree. However, `tool.exe` is the target of the linking rule defined by the `(executable)` stanza. This rule requires `main.cmx` and `config.cmx`. `main.cmx` is the target of the compilation rule for the `Main` module, which depends on `main.ml`. This file is in the source tree, so let's copy it under `_build`. This rule has all its dependencies available, so we can run its action, which writes `main.cmx`. Getting back to the dependencies of `tool.exe`, `config.cmx` is the target of the linking rule of the `Config` module. This rule has `config.ml` has a dependency. This file is itself the target of the configuration module rule, which lists `config.json` and `convert/json2ml.exe`. The first is available in the source tree and to simplify, let's assume that the second one has been built. This action has all its dependencies available, so we can execute its action to produce its target, `config.ml`. Now the module compilation rule for `Config` can be executed, producing `config.cmx`; and in turn the linking rule can be executed, producing `tool.exe`. Finally, `tool.exe` can be executed with `testdata.txt` as its argument.

In a nutshell: we recursively copied all the dependencies of the test rule, and executed the rules in the correct order.

This is a “cold build”, where there were no previous build artifacts. Note that if we change only part of the project (say the `main.ml` file), only a small number of rules will be evaluated, the ones that depend on `main.ml`.

### 5.1.5 Conclusion

Dune’s underlying model is based on rules. Stanzas are high-level constructs that can generate multiple rules, that are not always visible.

To build a target, Dune looks for the rule that produces that target and makes its way back to source files.

Rules define a directed acyclic graph which models dependency relations between files. Most of the rules in that graph may be executed for a cold build, but just the minimum will be executed for an incremental build.

## 5.2 Dune Projects and Workspaces

Whenever Dune builds anything, it does so at the level of a *Dune workspace*. A Dune workspace is a set of Dune projects. A typical workspace consists of a single Dune project.

A *Dune project* is defined by the presence of a *dune-project* file. Each Dune project extends over the file tree rooted at the directory containing the *dune-project* file, excluding any nested Dune projects.

Dune determines the root of the current workspace by finding the topmost ancestor containing a *dune-project* file or by the presence of a *dune-workspace* file (see *Finding the Root* and *Forcing the Root (for Scripts)* for details).

Different Dune projects within the same Dune workspace are independent of each other and no settings are shared between them, even if they are nested within each other.

Settings in *dune-workspace*, on the other hand, are inherited by all Dune projects in the workspace. Some settings (those that make sense for all projects) can be specified both in *dune-project* and *dune-workspace* files, with the former taking precedence. Note that all *dune-workspace* files other than the one specifying the root of the workspace are ignored.

Within a Dune project, *dune* files are used to define all objects of interest for Dune: libraries, executables, tests, etc. There are typically many *dune* files in a Dune project: one per directory, unless the directory does not contain anything relevant to Dune. In each *dune* file, references are resolved relative to the directory containing the file.

Note that there are specific stanzas and actions that may result in exceptions to some of the rules stated in the previous paragraph. See, for example, the *subdir* and *include\_subdirs* stanzas, as well as the *chdir* action.

Finally, note that only public items (public libraries, public executables) of a Dune project are visible to other Dune projects within the same Dune workspace.

## 5.3 How Preprocessing Works

Preprocessing consists in transforming source code before it is compiled. The goal of this document is to explain how this works in Dune.

Dune supports two separate ways of applying preprocessors, the “classic pipeline” (used with `(staged_pps)`), and the “fast pipeline” (used for all other *preprocessing specifications* including `(pps)`).

The OCaml compilers provide options for specifying a preprocessing step. The `-pp` option is used to invoke a textual preprocessor (something that reads text and returns text). The `-ppx` option is used to invoke a *ppx rewriter* (a function that takes an AST and outputs an AST).

This is the “classic pipeline”: preprocessing is part of the compilation itself. This is simple, but has a problem: in order to compute the dependencies of a module, it is necessary to pass the same `-pp` or `-ppx` option to `ocamldep`.

The classic pipeline has the following steps:

- preprocessing (as part of `ocamldep`)
- dependency analysis
- preprocessing (as part of compilation)
- compilation

Dune supports a “fast pipeline” where the preprocessor is invoked separately from the compiler and its output is saved. Afterwards the preprocessed code is compiled directly.

The fast pipeline has the following steps:

- preprocessing
- dependency analysis
- compilation

It has several advantages: it only invokes the preprocessor once per file, and the preprocessed code is reused between dependency analysis and different kinds of compilation. Also, when several preprocessors use `ppxlib`, they can be combined in a preprocessing program that traverses the AST only once.

However, some specific code generators or preprocessors require direct access to the compilation artefacts of their dependencies. Therefore they need to be used with the classic pipeline, even if it is slower. Note that a PPX is able to know if it was called as part of `ocamldep -ppx` or `ocamlopt -ppx`, so it can act differently in each phase.

Dune chooses which pipeline to use depending on the provided *Preprocessing Specification*. It will select the fast pipeline, unless (`staged_pps`) is used. In that case, the classic pipeline is used.

In the case of the fast pipeline, a single executable is built and accepts arguments for all preprocessors.

## 5.4 The OCaml Ecosystem

The OCaml ecosystem is not monolithic: the compiler and tools are not maintained by the same entities. As such, it can be difficult to understand the history and roles of the various pieces of this ecosystem. The goal of this page is to give a quick overview of the situation and the role that Dune plays in it.

### 5.4.1 The OCaml Compiler Distribution: Compiling and Linking

The [OCaml compiler distribution](#) contains “core” tools including the compilers (`ocamlc` and `ocamlopt`). They turn source files (with extensions `.ml` and `.mli`) into executables and libraries. Dependencies between compiled objects only exist at the module level, so this is a low-level tool.

### 5.4.2 Findlib: Metadata for Libraries

[Findlib](#) is a tool that defines the concept of library, so that libraries can depend on other libraries on top of the notion of module. Definitions of libraries, and other pieces of metadata, are stored in `META` files.

Findlib ships an executable named `ocamlfind` that can be used as a wrapper on top of the compilers to perform tasks such as producing an executable from compiled object files and external libraries.

### 5.4.3 Opam: a Collection of Software Projects

Opam is a package manager. It is used to determine which packages are necessary, and how to fetch and build them. Packages can contain libraries, executables, and other kinds of files.

The notion of version is specific to opam. If your project uses a function named `Png.read_file` but this function has been added only in version `1.2.0` of that package, opam needs to know about it.

Opam manages collections of installed packages, called switches. Using your project's dependencies (names and version constraints), it is able to create a switch that you'll be using to develop your project.

Public definitions of packages are available in a database called `opam-repository` which is maintained as a public Git repository. Publishing a package on opam (to make sure that external users can use your project) consists in adding its definition to `opam-repository`.

### 5.4.4 Dune: Giving Structure to Your Source Tree

Dune is a build system. It is used to orchestrate the compilation of source files into executables and libraries.

Assuming you have a development switch set up, you communicate to Dune about how your project is organized in terms of executables, libraries, and tests. It is then able to assemble the source files of your projects, with the dependencies installed in an opam switch, to create compiled assets for your project.

### 5.4.5 How Dune Integrates With the Ecosystem

Dune is designed to integrate with the tools mentioned above:

- By knowing how the OCaml compilers operate, it knows which build commands should be re-executed if some source files change.
- It outputs metadata like dependency information into META files that Findlib is able to make use of. This ensures that even if a project does not use Dune, it can use a library that has been produced by Dune. Conversely, it can read these files to determine dependency information for dependencies that have not been produced by Dune.
- It is able to generate opam files with filenames consistent with how opam looks for them. The generated files use build commands that make use of the `@install` and `@runtest` *aliases* so that the Dune abstractions map to the opam ones.

### 5.4.6 Dune is Opinionated

As described above, the OCaml ecosystem does not have a centralized toolchain. Units such as modules, libraries, and packages operate at different levels, and the relation between these can be confusing to users.

Dune tries to simplify the picture by reducing the difference between these objects:

- By default, a library will only expose a single top-level module named after the library (this is called a wrapped library).
- A library can only be installed in the package of the same name. This means that the names found in `dune-project` and `opam` files (package names) are consistent with the names found in `dune` files (library names). More precisely, libraries `foo`, `foo.bar` and `foo.baz` are part of the `foo` package.

## 5.5 How Package Management Works

This document explains how Dune's package management works under the hood. It requires a bit of familiarity with how opam repositories work and how Dune builds packages. Thus it is aimed at people who want to understand how the feature works, not how it is used.

For a tour on how to apply package management to a project, refer to the *OCaml Package Management With Dune* tutorial.

### 5.5.1 Motivation

A core part of modern programming is using existing code to save time. The OCaml package ecosystem has quite a long history with many projects building upon each other over many years. A significant step forward was the creation

of the OCaml Package Manager, `opam`, along with the establishment of a public package repository which made it a lot more feasible to share code between people and projects.

Over time, best practices have evolved, and while `opam` has incorporated some changes, it couldn't adopt all the modern workflows due to its existing user base and constraints.

Thus the Dune Package Management has been designed with a few core goals in mind:

- No global state visible to users, everything is local to projects
- Package management is configured through files (`dune-project` and optionally `dune-workspace`)
- Repositories are automatically kept up to date unless explicitly configured to use specific versions
- Builds can only access packages they have declared dependencies on
- Reproducible builds through lockfiles

Dune plays well with the existing OCaml ecosystem and does not introduce a new type of packages. Rather, it uses the same package repository and Dune packages stay installable with `opam`.

## 5.5.2 Package Management in a Project

This section describes what happens in a Dune project using the package management feature.

### Dependency Selection

The first step is to determine which packages need to be installed. Traditionally this has been defined in the `depends` field of a project's `opam` file(s).

Since version 1.10 Dune has supported *opam file generation* by specifying the package dependencies in the `dune-project`.

The package management feature uses the same metadata, as Dune will determine the list of packages to install from the `depends` field in the `dune-project` file. This allows projects to completely omit generation of `.opam` files, as long as they use Dune for package management. Thus all dependencies on OCaml packages are only declared in one single file.

To maintain compatibility with a large number of existing projects, Dune continues to support `.opam` files. While it is recommended to declare the dependencies directly in the `dune-project` file, it is not mandatory to do so. Dune will fall back to reading dependencies from `.opam` files when the package is not defined in `dune-project`.

### Locking

Given the list of the project's dependencies and their version constraints, the next steps are:

1. Find the transitive dependencies and figure out a version for each dependency that satisfies the constraints
2. For each dependency, download it, build it, and make it available to the project

In `opam`, `opam install` does both of these.

In Dune, these are separate steps: the first one is `dune pkg lock`, and the second one happens implicitly as part of *building*.

The idea of doing the first step and recording it for later is popular in other programming language package managers like NPM and is usually called locking. Creating a lock file ensures that the dependencies to be installed are always the same - unless that lock file is updated of course.

### **Note**

`opam` also supports creating lock files. However, these are not as central to the `opam` workflow as they are in the case of package management in Dune, which always requires a set of locked packages.

In the most general sense, a lock file is just a set of specific packages and their versions to be installed.

Instead of a lock file, Dune writes this information to a directory (the “lock directory”) with files that describe the dependencies. It includes the package’s name and version. Unlike many other package managers, the files include a lot of other information as well, such as the location of the source archives to download (since there is no central location for all archives), the build instructions (since each package can use its own way of building), and additional metadata like the system packages it depends upon.

The information is stored in a directory (`dune.lock` by default) as separate files, to reduce potential merge conflicts and simplify code review. Storing additional files like patches is also simpler this way.

## Package Repository Management

To find a valid solution that allows a project to be built, it is necessary to know what packages exist, what versions of these packages exist, and what other packages these depend on, etc.

In `opam`, this information is tracked in a central repository called `opam-repository`, which contains all the metadata for published packages.

It is managed using Git; `opam` typically uses a snapshot to find the dependencies when searching for a solution that satisfies the constraints.

Likewise, Dune uses the same repository; however, instead of snapshots of the contents, it uses the Git repository directly.

### **Note**

Dune maintains a shared internal cache containing all Git repositories that projects use. This way updates and checkouts are very fast because only new revisions have to be retrieved. The downside is that to be included in the cache, all the Git repos have to be cloned first which depending on the size of the repositories can take a bit of time.

On every call to `dune pkg lock`, Dune will update the metadata repository first (hence why efficiently updating that repository matters). This means that each `dune pkg lock` will use the newest set of packages available.

However, it is also possible to declare specific revisions of the repositories, to get a reproducible solution. Due to using Git, any previous revision of the repository can be used by specifying a commit hash.

Dune uses three repositories by default, in order of priority:

- `overlay` refers to `opam-overlays`, which defines packages patched to work with package management. The long-term goal is to have as few packages as possible in this repository as more and more packages work within Dune Package Management upstream. Check the *compatibility* section for details.
- `relocatable` refers to the `relocatable` branch of `dra27/opam-repository`, which provides a relocatable version of the OCaml compiler. This allows the compiler to be built and cached independently of the project’s build directory.
- `upstream` refers to the default branch of `opam-repository`, which contains all the publicly released packages.

## Solving

After Dune has read the constraints and loaded the set of candidate packages, it is necessary to determine which packages and versions should be selected for the package lock.

To do so, Dune uses `opam-0install-solver`, which is a variant of the `0install` solver to find solutions for opam packages.

Contrary to opam, the Dune solver always starts from a blank slate; it assumes nothing is installed and everything needs to be installed. This has the advantage that solving is now simpler, and previous solver solutions don't interfere with the current one. Thus, given the same inputs, it should always come up with the same result; no state is held between the solver runs.

This can lead to more packages being installed (as opam won't install new package versions by default if the existing versions satisfy the constraints), but it avoids interference from already installed packages that lead to potentially different solutions.

After solving is done, the solution gets written into the lock directory with all the metadata necessary to build and install the packages. From this point on, there is no need to access the package metadata repositories.

### Note

Solving and locking does not download the package sources. These are downloaded in the build step.

## Building

When building, Dune will read the information from the lock directory and set up rules for the packages. Check *The Dune Mental Model* for details about rules.

The rules that the package management sets up include:

- Fetch rules to download and unpack the source archives, and also download any additional sources such as patches
- Build rules to execute the build instructions stored in the lock directory
- Install rules to put the artifacts that were built into the appropriate Dune-managed folders

Creating these processes as rules mean that they will only be executed on demand, so if the project has already downloaded the sources, it does not need to download them again. Likewise, if packages are installed, they stay installed.

The results of the rules are stored in the project's `_build` directory and managed automatically by Dune. Thus, when cleaning the build directory, the installed packages are cleaned as well and will be reinstalled at the next build.

### 5.5.3 External System Dependencies (depexts)

OCaml packages sometimes require system packages at build or run time, such as C libraries, system tools, or `pkg-config` packages. In the opam ecosystem, these are called “depexts” (external dependencies) and are declared using the `depexts` field in opam files.

Dune does not install or manage system packages. It reads depexts information from opam package metadata and surfaces it to help users identify what needs to be installed on their system.

#### How Dune Handles Depexts

During dependency solving, Dune reads the `depexts` field from each package's opam metadata. The depexts and their platform filter conditions (using opam variables like `os-family`, `os-distribution`, and `arch`) are recorded in the lock directory. The filters are preserved as conditionals so that the lock directory remains portable across platforms. When depexts are queried or displayed (e.g., via `dune show depexts`), the filters are evaluated against the current platform.

### Surfacing Depexpts to Users

Dune surfaces depexpts information in two ways:

- **dune show depexpts** prints all system packages required by the project's locked dependencies (aggregated and deduplicated across all packages). See *How to Handle External System Dependencies (depexpts)* for usage.
- **Build failure hints:** when a locked package fails to build and has associated depexpts, Dune prints a hint listing that package's depexpts after the error message. This can help diagnose failures caused by missing system packages.

### 5.5.4 Packaging for Dune Compatibility

Dune can build and install most packages as dependencies, even if they are not built with Dune themselves. Dune will execute the build instructions from the lock directory, very similar to opam.

However, packages must adhere to certain rules to be compatible with Dune.

The most important one is that the packages must not use absolute paths to refer to files. That means they cannot read the path they are being built or installed in and expect this path to remain the same. Dune builds packages in a sandbox location, and after the build has finished, it moves the files to the actual destination.

#### Note

Unlike opam, Dune at the moment does not wrap the build in sandboxing tools like [Bubblewrap](#).

To comply with these restrictions the usual solution is to use relative paths, as Dune guarantees that packages installed into different sections are installed in a way where their relative location stays the same.

The `overlay` repository exists specifically to make currently non-compliant packages compatible with Dune's package management. It does so by supplying releases of packages where the current upstream releases don't support Dune package management yet.

## 5.6 How Dune integrates with opam

When instructed to do so (see *How to Generate Opam Files from dune-project*), Dune generates opam files with the following instructions:

```
build: [  
  ["dune" "subst"] {dev}  
  [  
    "dune"  
    "build"  
    "-p"  
    name  
    "-j"  
    jobs  
    "@install"  
    "@runtest" {with-test}  
    "@doc" {with-doc}  
  ]  
]
```

Let's see what this means in detail.

### 5.6.1 Substitution

The first step is to call `dune subst`, but only if the `{dev}` opam variable is set. This variable is only set when the package is pinned. This means that `dune subst` does not run for released versions, but it does for development versions.

This is not a problem since released versions should have a `(version)` field set in `dune-project`, and *placeholder substitution* should have been performed. `dune-release` takes care of these steps.

### 5.6.2 Opam Variables

In the second command line, `name` is a variable that evaluates to the name of the package being built, and `jobs` is a variable that corresponds to the number of commands to run in parallel.

### 5.6.3 What `-p` Means

The `-p` flag, shorthand for `--release-of-packages`, is Dune's public interface to set up the options for an opam build. The exact semantics may change, but as of Dune 3.8 it is equivalent to the combination of:

- `--root .`: set the *root* to prevent Dune from *looking it up*.
- `--only-packages name`: ignore packages other than `name` defined in the project.
- `--profile release`: set the *build profile* to `release`. In particular, this ensures that warnings are not fatal.
- `--ignore-promoted-rules`: silently ignores all rules with `(mode promote)`.
- `--default-target @install`: make sure that `dune build` with no target argument builds `@install`, not `@default` (this is not used in the opam integration since an explicit target is passed)
- `--no-config`: do not load the configuration file in the user's home directory.
- `--always-show-command-line`: ensures that the programs executed by Dune end up in the opam logs.
- `--promote-install-files`: ensures that `*.install` files are present in the source tree after the build.
- `--require-dune-project-file`: fail if `dune-project` is not present. In some previous Dune versions, `dune-project` could be generated when it is not present. This is not desirable with opam since the version the package has been prepared with is not known.

### 5.6.4 The Targets We're Building

The targets are specified as:

```
"@install"
"@runtest" {with-test}
"@doc" {with-doc}
```

The `{with- }` syntax is an opam filter. It means that the string before is present or not depending on the opam variable. These variables, in turn, are set depending on the opam configuration.

Concretely, in the next table, if the opam command on the left is executed, the Dune target on the right will be built:

opam command	Dune target
<code>opam install pkg</code>	<code>@install</code>
<code>opam install pkg --with-test</code>	<code>@install @runtest</code>
<code>opam install pkg --with-test --with-doc</code>	<code>@install @runtest @doc</code>

This filtering mechanism is also used to declare dependencies. If a package is using `lwt` and `alcotest`, but the latter only in its test suite, its `depends:` field is:

```
"lwt"  
"alcotest" {with-test}
```

This is expanded to just `lwt` in `opam install pkg`, but to `lwt alcotest` in `opam install pkg --with-test`.

The meaning of these *aliases* is the following:

- `@install` depends on all the `*.install` files in the project. In turn, these depend on all the installable files (libraries and executables with a public name and files that are manually installed through `(install)` stanzas).
- `@runtest` is the alias to which all tests are attached, including `(test)` stanzas. Building `@runtest` directly uses normal alias target semantics. See *Running Tests* for the command-line behavior of `dune runtest`.
- `@doc` executes `odoc` to create HTML docs under `_build`.

## 5.6.5 What Opam Expects From Dune

Given this `build:` lines and the fact that there is no `install:` line, what happens is the following:

- Opam executes `dune subst`, if the package is being pinned.
- Opam executes the build instruction, usually just `dune build -p pkg @install`
- This Dune command builds all the installable files and creates a `pkg.install` file.
- This file contains the paths to built files (somewhere in the `_build` directory) and the opam sections they should be installed in.
- Opam interprets this file and copies the built files to their destination. The install file is also used as a manifest of which files belong to which package, which is used when uninstalling the package.

## 5.7 How Dune Uses Dune to Build Dune

Dune's build system is itself Dune. This works thanks to a bootstrap process. This document explains how this works.

### 5.7.1 boot/bootstrap.ml

`boot/bootstrap.ml` is an OCaml script (it is interpreted, not compiled) that is a mini-build system tailored to Dune itself. It computes dependencies between the various modules by calling `ocamldep`, and it will generate build and link commands. It knows how to execute these commands in parallel. It does not read any dune file. However, the project structure and its system dependencies are encoded in `boot/libs.ml`.

This step produces `_boot/dune.exe`.

### 5.7.2 Completing the Opam Installation

`_boot/dune.exe` is the bootstrap Dune. Since it has been built from the Dune sources, it will act like Dune: it can read dune files, etc.

This is actually the dune executable that will get installed. This is the same executable as the one obtained by running `opam install dune`. At this stage of the process, Opam does not know about this: it expects a `dune.install` file that explains what files to install.

The next command run by the Opam instruction is the following:

```
$ ./_boot/dune.exe build dune.install --release --profile dune-bootstrap
```

By using the `dune-bootstrap` *build profile*, it does not run a full build, but only copy `_boot/dune.exe` to its install location (as the `dune` binary), and generate `dune.install`.

### 5.7.3 make dev: Everything Else for Local Development

The above describes how Dune itself is built through Opam, but that's not all there is to it: the Dune repository contains other libraries that need to be built. In fact, executing the `boot/bootstrap.ml` script did not generate files useful for editor integration.

So the main Makefile has a `make dev` target that will run `_boot/dune.exe build @install`: this will rebuild the project using Dune itself.

As a special rule, this build will regenerate `boot/libs.ml` using the locations of the internal libraries used to build Dune.

## 5.8 A Tour of the Dune Codebase

### Note

This document is based on Dune 3.15.0, whose source can be browsed [here](#). The links in this tour point to this version, but will not reflect how this works in other versions of Dune.

Let's start with a very high level tour of how `dune build` operates.

As explained in *The Dune Mental Model*, `dune build` will interpret the targets listed on the command line, interpret the dune files in the workspace as rules, and execute the rules relevant to the requested targets.

These steps correspond to areas of the Dune codebase:

- the command-line interface is defined in `bin/`;
- the dune files are interpreted using a library defined in `src/dune_rules/`;
- they are registered into an engine in `src/dune_engine/`.

Next, we will go deeper into these areas.

### 5.8.1 Command-Line Interface

The command-line interface is defined using `cmdliner`. One thing to note is that we use binding operators to compose terms:

#### `bin/print_rules.ml`

```

174 let+ builder = Common.Builder.term
175 and+ out =
176   Arg.(
177     value
178     & opt (some string) None
179     & info [ "o" ] ~docv:"FILE" ~doc:"Output to a file instead of stdout.")
180 and+ recursive =
181   Arg.(
182     value
183     & flag
184     & info
185       [ "r"; "recursive" ]
186     ~doc:
187       "Print all rules needed to build the transitive dependencies of the given_

```

(continues on next page)

```

188     targets.")
189 and+ syntax = Syntax.term
190 and+ targets = Arg.(value & pos_all dep [] & Arg.info [] ~docv:"TARGET") in

```

## 5.8.2 Parsing of Dune Files

Parsing dune files is done in two steps:

- They are parsed as S-expressions using `src/dune_sexp/parser.mli`;
- Then they are decoded using `src/dune_sexp/decoder.mli`. The result of this decoding step is added to an extensible variant using a mechanism in `src/dune_lang/stanza.mli`.

Instead of writing a parser or using pattern matching, we define decoders, which are abstract values of type `'a Decoder.t` (returning a value of type `'a`). These decoders are assembled using combinators. For example, we can use simple decoders to write a decoder for a record type. This decoder abstraction is monadic, but the applicative subset is sufficient for most decoders.

As an example, here is how `(copy_files)` is parsed:

`src/dune_rules/stanzas/copy_files.ml`

```

31 let long_form =
32   let check = Dune_lang.Syntax.since Stanza.syntax (2, 7) in
33   let+ alias = field_o "alias" (check >>> Dune_lang.Alias.decode)
34   and+ mode = field "mode" ~default:Rule.Mode.Standard (check >>> Rule_mode_decoder.
  ↳ decode)
35   and+ enabled_if = Enabled_if.decode ~allowed_vars:Any ~since:(Some (2, 8)) ()
36   and+ files = field "files" (check >>> String_with_vars.decode)
37   and+ only_sources =
38     field_o
39       "only_sources"
40       (Dune_lang.Syntax.since Stanza.syntax (3, 14) >>> decode_only_sources)
41   and+ syntax_version = Dune_lang.Syntax.get_exn Stanza.syntax in
42   let only_sources = Option.value only_sources ~default:Blang.false_ in
43   { add_line_directive = false
44     ; alias
45     ; mode
46     ; enabled_if
47     ; files
48     ; only_sources
49     ; syntax_version
50   }

```

The fields are queried individually, and a record is built using all the intermediate results. This will automatically take care of generating “unknown field X,” “duplicate field X,” and similar error messages.

Another interesting thing to note is that the fields are not decoded directly, but use the following pattern:

```
Syntax.since Stanza.syntax (x, y) >>> decoder
```

Let’s unpack this: (`>>>`) will run a unit `Decoder.t` on the input before passing the input to an actual decoder. The first decoder can be used to implement a check and trigger an error in some cases.

Here, it is used for versioning. For example the `(copy_files)` stanza started supporting `(enabled_if)` in version 2.8. Decoding this field is protected by this `since` call: it means that if the language version in `dune-project` file is

greater than 2.8. In particular, this ensures that the project can not be built with Dune versions older than 2.8.0.

Once decoding succeeds, various stanzas are turned into various types defined in `src/dune_rules/stanzas/`.

### 5.8.3 Rule Generation

Using these parsed stanzas, the next step is to generate rules. This work starts in `src/dune_rules/gen_rules.ml`, which dispatches to various modules in `src/dune_rules/`.

Rules are registered on the build engine using the following function from the `Super_context` module:

```
val add_rule
  : t
  -> ?mode:Rule.Mode.t
  -> ?loc:Loc.t
  -> dir:Path.Build.t
  -> Action.Full.t Action_builder.With_targets.t
  -> unit Memo.t
```

A value of `Super_context.t` represents an OCaml toolchain (`Context.t`) as well as various capabilities to expand variables and refer to (*env*) *stanzas*. The last, unlabelled argument corresponds to the fully annotated action. We'll go through its type below.

The modules in `src/dune_rules` often expose a function `gen_rules` taking a parsed stanza, a `Super_context.t` value, a directory name (and other arguments), and returning `unit Memo.t`.

#### Note

The `Memo` module is central to how Dune operates. It is a monadic memoization framework that allows two things:

- Sharing and caching expensive internal computations, such as computing the list of libraries Dune knows about, or computing the list of flags that should be used to compile a given module.
- Incremental recomputation of this cached data. `Memo` tracks dependencies between memoized values and will only recompute the necessary ones when an input changes. This is a mini in-memory build system that works like a spreadsheet. It is essential to the watch mode.

An example rule is the *mdx* stanza, implemented in `src/dune_rules/mdx.ml`. There are several steps in setting up rules and alias actions for a (*mdx*) stanza:

- Run `ocaml-mdx deps` on the input file to discover extra dependencies.
- Run `ocaml-mdx dune-gen` to produce a `mdx_gen.ml-gen` OCaml source file.
- Compile this generated executable.
- Attach an action to `@runtest` that creates the `.corrected` file and immediately compares it with the original file using `diff`.

Let's walk through these pieces.

The rule that generates `mdx_gen.ml-gen` is a regular rule made by running a command:

```
Command.run_dyn_prog
  ~dir:(Path.build dir)
  mdx_prog
  ~stdout_to
  [ A "dune-gen"
```

(continues on next page)

(continued from previous page)

```

; prelude_args
; Resolve.Memo.args directory_args
; Lazy.force color_always
]

```

Here `mdx_prog` is the resolved path to `ocaml-mdx`. It can point to a binary in `PATH` or to a binary built in the current workspace. The command arguments are written in a domain-specific language defined in `src/dune_rules/command.mli`. For example, `A` is a plain string, and `Path` or `Dep` arguments let Dune track paths used by the command. The `~stdout_to` parameter tells Dune which target this rule produces.

The generated executable is then built with `Exe.build_and_link`.

The dependencies reported by `ocaml-mdx deps` are not stored in a separate `.mdx.deps` target. Instead, Dune runs that command as an anonymous action and reads its standard output:

```

let read ~sctx ~dir ~loc ~mdx_prog (files : Files.t) =
  let open Action_builder.0 in
  (let* prog = mdx_prog in
   Command.run'
     ~dir:(Path.build dir)
     prog
     [ Command.Args.A "deps"
       ; Lazy.force color_always
       ; Dep (Path.build files.src)
     ])
  |> Super_context.execute_action_stdout sctx ~loc ~dir
  |> Action_builder.of_memo
  >>| parse

```

`Super_context.execute_action_stdout` executes the command as an anonymous action and returns its captured standard output to the rule generator. The result is parsed into the dynamic dependencies of the `mdx` test action.

Finally, Dune attaches one anonymous action to the `@runtest` alias. This action runs `mdx` to produce the `.corrected` file and then runs the `diff` action that compares it with the source file:

```

let mdx_action =
  let mdx_input_dependencies = (* Deps.read ... *) in
  let executable, command_line, redirect_stdout = (* ... *) in
  let env, sandbox = (* ... *) in
  let open Action_builder.0 in
  let action =
    Action_builder.env_var "MDX_RUN_NON_DETERMINISTIC"
    >>> (Action_builder.map mdx_input_dependencies ~f:(fun d -> ()), d)
    |> Action_builder.dyn_deps)
    >>> let* executable = executable in
      Command.run'
        ~dir:(Path.build dir)
        ~env
        ~sandbox
        executable
        command_line
  in
  let action =
    if redirect_stdout

```

(continues on next page)

(continued from previous page)

```

then
  Action_builder.map
    action
    ~f:(Action.Full.map ~f:(Action.with_stdout_to files.corrected))
  else action
in
let+ action
and+ locks = Expander.expand_locks expander stanza.locks in
let mkdir_corrected_dir =
  Action.mkdir (Path.Build.parent_exn files.corrected) |> Action.Full.make
in
let diff =
  Action.diff ~optional:false (Path.build files.src) files.corrected
  |> Action.Full.make
in
Action.Full.reduce
  [ mkdir_corrected_dir; Action.Full.add_locks locks action; diff ]
in
Super_context.add_alias_action
  sctx
  (Alias.make Alias0.runtest ~dir)
  mdx_action
  ~loc
  ~dir

```

Here, `Action_builder.dyn_deps` makes the dependencies discovered by `ocaml-mdx_deps` part of the anonymous action. The `.corrected` file is an intermediate file used by the same alias action, not a target with its own standalone build rule. If the diff finds a mismatch, it registers the usual promotion from the corrected file back to the source file.

### Note

`Action_builder` is different from `Memo`, which corresponds to what happens within Dune itself. But it is also possible to use `Memo` from an `Action_builder` context. In that sense, `Action_builder` is more powerful: at execution time, `Action_builder` will manage what happens in the `_build` directory, while `Memo` is only concerned with what happens in memory.

## 5.8.4 The Engine

The engine is the core, reusable part of Dune. It contains all the composable primitives that make it a build system.

The fact that it is split from the *rules part* makes it possible to create a different build system using this library. For example, Jane Street internally uses a build system with this engine as a backend, but a different frontend and CLI.

In the context of Dune, the engine keeps track of the various directories and the rules in them and is able to build files using them. In addition, it takes care of the various caches that Dune uses, such as the one present in the `_build` directory, the *shared cache*, etc.

## 5.8.5 Libraries

Dune, as a package, is primarily an executable, but its source tree embeds a few public libraries. These are developed in `otherlibs/`.

Some of these have a special link to Dune, such as `dune-build-info` or `dune-site`. Others are just helper libraries

that we develop as part of Dune, but they have a strong relation to the Dune internals, like `dyn` or `xdg`.

 **See also**

*Dune Libraries*

### 5.8.6 Vendored Libraries

As an opam package, Dune has no dependencies. But it uses some existing libraries by copying, or “vendoring”, their source code into the `vendor/` directory.

In some cases, the external dependency is extracted from the upstream repository. In other cases, we carry patches and refer to a fork in the [ocaml-dune GitHub organization](#).

The source code in the `vendor/` directory is not meant to be edited directly. Instead, it is edited in the external repository, and the copy in the Dune source tree is updated by running an update script, such as `update-spawn.sh`.

### 5.8.7 Tests

The `test/` directory contains all the tests for Dune itself. Additionally, the tests for our *Libraries* are stored in `otherlibs/` next to the library itself.

We have 3 kind of tests:

- Unit tests, in `test/unit-tests` (we have very few of these, usually preferring other kinds)
- Expect tests, in `test/expect-tests` (using `ppx_expect`)
- *Cram tests*, in `test/blackbox-tests/`. This is our preferred way of testing.

The actual Cram tests are in `test/expect-tests/test-cases`. There is a mix of file tests and directory tests. For regression tests, the pattern `githubNUMBER.t` is used.

The `dune` file at `test/expect-tests/test-cases/dune` sets up some metadata for the tests. For example, if a test has an external dependency like `strace`, a dependency on `%{bin:strace}` will prevent the test from even trying to start. Some tests are also disabled on some configurations using `(enabled_if)`.

Finally, some programs available in the Cram tests are defined in `test/expect-tests/blackbox-tests/utils`. For example, we have a `dune_cmd` program that contains reimplementations of common utilities like `stat`, which do not have the same output on the different systems we use to test Dune.

 **See also**

*Working on the Dune Codebase*

## ADVANCED TOPICS

These documents describe some advanced or very specific features of Dune.

### 6.1 Dynamic Loading of Packages with Findlib

The preferred way for new development is to use *Plugins and Dynamic Loading of Packages*.

Dune supports the `findlib.dynload` package from [Findlib](#) that enables dynamically-loading packages and their dependencies (using the OCaml Dynlink module). Adding the ability for an application to have plugins just requires adding `findlib.dynload` to the set of library dependencies:

```
(library
  (name mytool)
  (public_name mytool)
  (modules ...))

(executable
  (name main)
  (public_name mytool)
  (libraries mytool findlib.dynload)
  (modules ...))
```

Use `F1_dynload.load_packages l` in your application to load the list `l` of packages. The packages are loaded only once, so trying to load a package statically linked does nothing.

A plugin creator just needs to link to your library:

```
(library
  (name mytool_plugin_a)
  (public_name mytool-plugin-a)
  (libraries mytool))
```

For clarity, choose a naming convention. For example, all the plugins of `mytool` should start with `mytool-plugin-`. You can automatically load all the plugins installed for your tool by listing the existing packages:

```
let () = Findlib.init ()
let () =
  let pkgs = F1_package_base.list_packages () in
  let pkgs =
```

(continues on next page)

```
List.filter
  (fun pkg -> 14 <= String.length pkg && String.sub pkg 0 14 = "mytool-plugin-")
  pkgs
in
Fl_dynload.load_packages pkgs
```

## 6.2 Profiling Dune

Dune writes detailed trace data about internal operations (such as command timing) to `_build/trace.csexp` by default. Use `--trace-file FILE` to write to a different location.

To load traces into Chromium's `chrome://tracing` or `Perfetto`, convert them to Chrome trace format:

```
$ dune trace cat --chrome-trace > trace.json
```

## 6.3 Package Version

Dune records package versions in generated metadata such as generated opam files, `META` files, and `dune-package` files. The same version is available in Dune files with the `%{version:<package>}` variable.

For packages declared with a `package` stanza, Dune determines the version in this order:

1. The package-specific `version` field in the `(package ...)` stanza.
2. The top-level `version` field in `dune-project`.
3. No version.

When a package is declared in `dune-project`, Dune does not read the version from an existing `<package>.opam` file for generated metadata. If you want Dune to manage opam files, use `generate_opam_files` and keep the version in `dune-project`.

For projects without any `(package ...)` stanzas, Dune can infer packages from existing opam files. In that case, the version from `<package>.opam` is used first; if the opam file has no version, Dune falls back to the top-level `(version ...)` field in `dune-project`.

Ordinary builds do not use the VCS version as a fallback for generated package metadata. VCS-derived versions are used by `dune subst` and by the `dune-build-info` library. The way Dune obtains the version from the VCS is described in *the build-info section*.

## 6.4 OCaml Syntax

If a dune file starts with `(* -*- tuareg -*- *)`, then it is interpreted as an OCaml script that generates the dune file as described in the rest of this section. The code in the script will have access to a `Jbuild_plugin` module containing details about the build context it's executed in.

The OCaml syntax gives you an escape hatch for when the S-expression syntax is not enough. It isn't clear whether the OCaml syntax will be supported in the long term, as it doesn't work well with incremental builds. It is possible that it will be replaced by just an `include` stanza where one can include a generated file.

Consequently **you must not** build complex systems based on it.

## 6.5 Variables for Artifacts

For specific situations where one needs to refer to individual compilation artifacts, special variables (see *Variables*) are provided, so the user doesn't need to be aware of the particular naming conventions or directory layout implemented by Dune.

These variables can appear wherever a *Dependency Specification* is expected and also inside *Actions*. When used inside *Actions*, they implicitly declare a dependency on the corresponding artifact.

The variables have the form `%{<ext>:<path>}`, where `<path>` is interpreted relative to the current directory:

- `cmo:<path>`, `cmx:<path>`, and `cmi:<path>` expand to the corresponding artifact's path for the module specified by `<path>`. The basename of `<path>` should be the name of a module as specified in a `(modules)` field.
- `cma:<path>` and `cmxa:<path>` expands to the corresponding artifact's path for the library specified by `<path>`. The basename of `<path>` should be the name of the library as specified in the `(name)` field of a `library` stanza (*not* its public name).
- `cmt:<path>` and `cmti:<path>` expand to the corresponding compiled annotation files for the module specified by `<path>`. These files contain the typed abstract syntax tree with precise location information and type annotations, generated with the `-bin-annot` flag. They are particularly useful for IDE tools to provide tooltips and type information.

Added in version 3.21.

In each case, the expansion of the variable is a path pointing inside the build context (i.e., `_build/<context>`).

## 6.6 Building an Ad Hoc `.cmxs`

In the model exposed by Dune, a `.cmxs` target is created for each library. However, the `.cmxs` format itself is more flexible and is capable to containing arbitrary `.cmxa` and `.cmx` files.

For the specific cases where this extra flexibility is needed, one can use *Variables for Artifacts* to write explicit rules to build `.cmxs` files not associated to any library.

Below is an example where we build `my.cmxs` containing `foo.cmx` and `d.cmx`. Note how we use a *library* stanza to set up the compilation of `d.cmx`.

```
(library
 (name foo)
 (modules a b c))

(library
 (name dummy)
 (modules d))

(rule
 (targets my.cmxs)
 (action (run %{ocamlc} -shared -o %{targets} %{cmxa:foo} %{cmx:d})))
```



## MISCELLANEOUS

These documents contain tidbits of info that do not fit anywhere else, and information about the project itself.

### 7.1 FAQ

#### 7.1.1 Why do many Dune projects contain a Makefile?

Many Dune projects contain a root `Makefile`. It's often only there for convenience for the following reasons:

1. There are many different build systems out there, all with a different CLI. If you have been hacking for a long time, the one true invocation you know is `make && make install`, possibly preceded by `./configure`.
2. You often have a few common operations that aren't part of the build, so `make <blah>` is a good way to provide them.
3. `make` is shorter to type than `dune build @install`

#### 7.1.2 How to add a configure step to a Dune project?

The `with-configure-step` example shows one way to add a configure step that preserves composability; i.e., it doesn't require manually running the `./configure` script when working on multiple projects simultaneously.

#### 7.1.3 Can I use `topkg` with Dune?

While it's possible to use the `topkg-jbuilder`, it's not recommended. `dune-release` subsumes `topkg-jbuilder` and is specifically tailored to Dune projects.

#### 7.1.4 How do I publish my packages with Dune?

Dune is just a build system and considers publishing outside of its scope. However, the `dune-release` project is specifically designed for releasing Dune projects to opam. We recommend using this tool for publishing Dune packages.

#### 7.1.5 Where can I find some examples of projects using Dune?

The `dune-universe` repository contains a snapshot of the latest versions of all opam packages that depend on Dune. Therefore, it's a useful reference to find different approaches for constructing build rules.

#### 7.1.6 What is Jenga?

`jenga` is a build system developed by Jane Street, mainly for internal use. It was never usable outside of Jane Street, so it's not recommended for general use. It has no relationship to Dune apart from Dune being the successor to Jenga externally. Eventually, Dune is expected to replace Jenga internally at Jane Street as well.

### 7.1.7 How to make warnings non-fatal?

*jbuilder* formerly displayed warnings, but most of them wouldn't stop the build. However, Dune makes all warnings fatal by default. This can be a challenge when porting a codebase to Dune. There are two ways to make warnings non-fatal:

- The *jbuilder* compatibility executable works even with *dune* files. You can use it while some warnings remain and then switch over to the *dune* executable. This is the recommended way to handle the situation.
- You can pass `--profile release` to *dune*. It will set up different compilation options that usually make sense for release builds, including making warnings non-fatal. This is done by default when installing packages from *opam*. See *profile* for the exact defaults associated with each standard profile.
- You can change the flags used by the *dev* profile by adding the following stanza to a *dune* file:

```
(env
 (dev
  (flags (:standard -warn-error -A))))
```

### 7.1.8 How to turn specific errors into warnings?

Dune is strict about warnings by default in that all warnings are treated as fatal errors. To change certain errors into warnings for a project, you can add the following to *dune-workspace*:

```
(env (dev (flags :standard -warn-error -27-32)))
```

In this example, the warnings 27 (*unused-var-strict*) and 32 (*unused-value-declaration*) are treated as warnings rather than errors.

### 7.1.9 How to display the output of commands as they run?

When Dune runs external commands, it redirects and saves their output, then displays it when complete. This ensures that there's no interleaving when writing to the console.

But this might not be what you want. For example, when you debug a hanging build or need to inspect a large command output without truncation.

In that case, one can pass `-j1 --no-buffer` so the commands are directly printed on the console (and the parallelism is disabled so the output stays readable). This also avoids Dune's buffering of command output, so redirecting Dune's output to a file can be used to capture the full, untruncated output. If the captured output contains terminal colour escape sequences, set `CLICOLOR=0` in the environment to disable colours.

### 7.1.10 How can I generate an *mli* file from an *ml* file?

When a module starts as just an implementation (*.ml* file), it can be tedious to define the corresponding interface (*.mli* file).

It is possible to use the `ocaml-print-intf` program (available on *opam* through `$ opam install ocaml-print-intf`) to generate the right *mli* file:

```
$ dune exec -- ocaml-print-intf ocaml_print_intf.ml
val root_from_verbos_output : string list -> string
val target_from_verbos_output : string list -> string
val build_cmi : string -> string
val print_intf : string -> unit
val version : unit -> string
val usage : unit -> unit
```

The `ocaml-print-intf` program has special support for Dune, so it will automatically understand external dependencies.

### 7.1.11 How can I build a single library?

You might want to do this when you don't have all the dependencies installed to compile an entire project, or parts of the project don't build for whatever reason. Maybe you want to check if your changes compile or produce build artifacts needed by `ocaml-lsp-server`.

Suppose you have a library defined in `src/foo/dune`:

```
(library
 (public_name my_library)
 ...)
```

You can build this library on its own by running the following from the project root directory:

```
$ dune build %cmxa:src/foo/my_library}
```

Note that the path (`src/foo` in the example above) is relative to the current directory - not the project root. If the library defines a name distinct from its `public_name` then that can be used interchangeably with the `public_name` in this command.

### 7.1.12 Why does `source_tree` ignore files and directories when they begin with a “.” (period)?

Dune's default behaviour is to ignore files and directories starting with “.” when copying directories with `source_tree`. This is to avoid accidentally copying the `.git` directory into the `_build` directory during a build.

This is a common source of confusion when interoperating with other libraries that use hidden directories for configuration, such as Rust. For example consider this rule which builds a Rust library contained in a subdirectory `foo-rs`:

```
(rule
 (target foo.a)
 (deps
 (source_tree foo-rs))
 (action
 (progn
 (chdir
 foo-rs
 (run cargo build --release))
 (run mv foo-rs/target/release/%{target} %{target}))))))
```

The build config for the Rust project will be in a directory `foo-rs/.cargo/config.toml`, and by default the `.cargo` directory won't get copied into the `_build` directory and so the Rust project will build with an incorrect configuration.

To fix this, create a `dune` file at the top level of the Rust project (i.e., `foo-rs/dune`):

```
(dirs :standard .cargo)
```

If you're following the standard advice for embedding Rust projects into Dune projects then you likely already have a `dune` project inside your Rust project that looks like:

```
(dirs :standard \ target)
(data_only_dirs vendor)
```

In this case you can update it to look like this:

```
(dirs :standard .cargo \ target)
(data_only_dirs vendor)
```

### 7.1.13 Why can't I write inline tests in a package without users needing to install `ppx_inline_test`?

If you came to OCaml from Rust and noticed that Dune has a feature for running inline tests you might be wondering how to do the OCaml equivalent of:

```
// define a private function
fn foo() { ... }

// test the function right next to its definition
#[test]
fn test_of_foo() { ... }
```

That is, writing tests for private functions right next to the definition of those functions. The *Inline Tests* documentation describes how to do this using the `ppx_inline_test` package; however, if you do this in your package, then your package must *unconditionally* depend on the `ppx_inline_test` package. Opam has a notion of test-only dependencies (its `with-test` flag), but you cannot use this with `ppx_inline_test`. The consequence of this is that anyone depending on your package is also transitively depending on `ppx_inline_test` as well as all of its dependencies.

The reason for this is OCaml code with preprocessor directives (such as those used for inline tests with `ppx_inline_test`) is technically not valid OCaml code until it has been preprocessed. Unlike the cargo build system used for Rust, Dune does not have a preprocessor built into it. Instead, it relies on external tools (such as `ppx_inline_test`) to parse the code and replace any preprocessor directives with valid OCaml. Dune doesn't know how to parse OCaml code at all so it can't even remove inline tests from the code in cases where `ppx_inline_test` is unavailable.

The blessed workaround for folks who want to use `ppx_inline_test` in their packages but don't want to add it as a dependency is to create a new (unreleased) package which contains all the tests. In the original package, expose all the private APIs you intend to test via public modules named something foreboding such as `For_test` so your users know not to rely on their contents and then have the test package define tests that call your "private" APIs through the `For_test` modules.

## 7.2 Goal of Dune

The Dune project strives to provide the best possible build tool for the entire OCaml community, including individual developers contributing to open source projects in their free time, larger companies (such as Jane Street), and communities, like MirageOS and Irmin. Additionally, we aim to provide the same features for other neighbouring communities, such as Rocq and possibly Reason/Bucklescript, in the future.

We haven't reached this goal yet, as Dune still requires development in some areas to be such a tool, but we're steadily working towards that goal. On a practical level, a few boxes must be checked, and a considerable number of details needs to be sorted out. At a high-level, we think a tool that works for everyone in the OCaml community should at least:

1. have excellent backward compatibility properties
2. have a robust and scalable core
3. remain a no-brainer dependency
4. remain accessible
5. have very good support for the OCaml language

6. be extensible

At this point, we've done a good job at 1, 3, 4, and 5. We're currently working towards 2 and are doing the preparatory work for 6. Once all these boxes have been checked, we'll consider the Dune project complete.

Below, we develop each point and give some insights into our current and future focuses.

### 7.2.1 Have Excellent Backward-Compatibility Properties

In an open source community, two types of groups exist: those with enough resources to continuously bring their projects up-to-date and those who work on them in their free time. The latter obviously can't provide the same level of continuous support and updates as the former.

From the Dune point of view, we consider every released project with dune files a precious piece that will potentially never change, so we discourage changing Dune in a way where it could no longer understand a released project.

Of course, we can't give a 100% guarantee that Dune will always behave exactly the same. That would be unrealistic and would prevent the project from moving forward. In order to provide good backward-compatibility properties while still keeping the project fresh and dynamic, we need to properly delimit, document, and version the set of behaviours on which users rely. For this to be manageable, the surface Dune API must remain small.

A distinguishing feature of Dune allows the user to declare which version of the dune tool they wrote the project against, and dune will morph itself to behave the same as this version of the dune binary, even if it's a newer version. As a result, a recent dune binary version can understand a wide range of Dune projects written against many different versions of Dune, and while we strictly follow [semantic versioning](#), new major versions of Dune effectively introduce very few breaking changes. Most projects don't need upper bounds on Dune.

This guarantee is of course limited to documented behaviours.

### 7.2.2 Have a Robust and Scalable Core

Tech companies tend to be fond of big mono repositories, so for compatibility, Dune must consume large repositories without blinking. It not only needs to build fast, but more importantly, it must not impede fast feedback during development, no matter the size of the repository.

Note that we'll only test Dune on repositories as large as people participating in Dune's development require. Currently, the largest user is Jane Street. If someone wanted to use Dune on much larger repositories than the ones used at Jane Street, and this required a significant amount of effort on Dune, this wouldn't be considered unless we get some help to do so and we can keep the other promises.

In particular, while making Dune scalable, we must also ensure Dune doesn't turn into a monster, because no one wants to force their users to install a monster to build their project. This brings us to the next point of Dune being a no-brainer dependency:

### 7.2.3 Remain a No-Brainer Dependency

Dune is a hard dependency of any Dune project. Anyone using Dune to develop their project will have to ask their user to install Dune. For this reason, it is very important to keep Dune as lean as possible.

We need to be careful when we start relying on an external piece of software or when we introduce new concepts. We must not introduce duplication or useless stuff. The overall projects has to remain lean.

It's also important to keep Dune as easy to install as possible. Currently, the only requirement to build Dune is a working OCaml compiler. Nothing else is required, not even a shell, and we should keep it this way.

### 7.2.4 Remain Accessible

Since Dune aims to be the best possible tool for the whole OCaml community, it's important to keep Dune accessible. Getting started and learning Dune should be straightforward.

For that purpose, when designing the language (the command line interface or the documentation), we must take on the new-user perspective, one who just discovered Dune and its features, because Dune should be suitable for everyone! It also needs to provide advanced and more complex features for expert users. However, the documentation should always flow from the simpler concepts and common tasks to the more complex ones, even if the simpler features can be explained as instances of the more general ones.

### 7.2.5 Have Excellent Support for the OCaml Language

There are many, many build systems out there. Dune stands out because it primarily targets the OCaml community, so Dune must come with excellent support for the OCaml language and OCaml projects in general.

If it didn't, Dune would just be yet another generic build system.

Perhaps in the future some of the general build system will take over, and Dune might just become a plugin in this system. It could even disappear into the language, if the compiler gains significant high-level features. But for now, Dune is a standalone build system that primarily serves the OCaml community's needs, and to the extent that is reasonably possible, the needs of other functional language communities.

### 7.2.6 Be Extensible

No matter the quality of the OCaml language's support, it will never be enough to cover every single project need. For this reason, Dune must provide some form of openness for projects with need that don't completely fit in the Dune model.

In the long run, extensibility tends to obstruct innovation, and we should always strive to ensure that we cover all the general needs of the main Dune language; however, we'll always need an escape hatch for Dune to remain a practical choice.

It's pretty clear that extensibility must be done via OCaml code, and currently it's a bit difficult to use OCaml as a proper extension language, though some work is being done to help on that front.

## 7.3 Working on the Dune Codebase

This section gives guidelines for working on Dune itself. Many of these are general guidelines specific to Dune. However, given that Dune is a large project developed by many different people, it's important to follow these guidelines in order to keep the project in a good state and pleasant to work on for everybody.

### 7.3.1 Dependencies

To create a directory-local opam switch with the dependencies necessary to build the tests, run:

```
$ make dev-switch
```

This can also be used to keep the switch updated when dependencies change.

The Makefile also has a `make dev-deps` which will install just the dependencies used by tests. These are marked `{ with-dev-setup }` in Dune's opam file.

### 7.3.2 Bootstrapping

Dune uses Dune as its build system, which requires some specific commands to work. Running `make dev` bootstraps (if necessary) and runs `./dune.exe build @install`.

If you want to just run the bootstrapping step itself, build the `bootstrap` phony target with

```
$ make bootstrap
```

You can always rerun this to bootstrap again.

Once you've bootstrapped Dune, you should be using it to develop Dune itself. Here are the most common commands you'll be running:

```
# to make sure everything compiles:
$ ./dune.exe build @check
# run all the tests
$ ./dune.exe runtest
# run a particular cram foo.t:
$ ./dune.exe build @foo
```

#### See also

*[How Dune Uses Dune to Build Dune](#)*

### 7.3.3 Writing Tests

Dune uses expectation-style tests for both unit tests and integration tests. For unit tests, we use the `ppx_expect` framework, where we introduce tests via `let%expect_test`, and where we use `[%expect ...]` nodes to capture expectations:

```
let%expect_test "<test name>" =
  print_string "Hello, world!";
  [%expect {|
    Hello, world!
  |}]
```

For integration tests, we use a system similar to [Cram tests](#) for testing shell commands and their behavior.

Test the output of `echo`:

```
$ echo 'Hello, world!'
Hello, world!
```

Test a nonzero exit status:

```
$ false
[1]
```

Test a multi-line shell invocation:

```
$ cat <<EOF
> multi
> line
> EOF
```

(continues on next page)

```
multi
line
```

Cram tests run with `sh` by default. Use only portable shell syntax unless a *Cram* stanza selects (`shell bash`) for the test.

These tests must be reproducible, so it is often necessary to filter command output to show only relevant parts. This also prevents tests from breaking due to unrelated changes and makes them easier to read.

For tests that run on multiple platforms, use only commands available everywhere. When platform-specific functionality is needed, Cram tests can use `dune_cmd`, an OCaml tool that provides portable implementations of common operations: file statistics, waiting for files to appear, waiting for filesystem clocks to advance, and a subset of `sed` features with clearer syntax.

The `censor` helper replaces hex digests and other values subject to change with stable labels. Distinct digests get distinct labels:

```
$ echo paths | censor
_build/.sandbox/$DIGEST1/foo.txt
_build/default/.ppx/$DIGEST2/ppx.exe
```

### ➔ See also

#### timer tests

An example of expect-tests.

#### Cram test examples

A collection of Cram tests.

When running Dune inside tests, the `INSIDE_DUNE` environment variable is set. This has the following effects:

- Change the default root detection behaviour to use the current directory rather than the top most `dune-project` / `dune-workspace` file.
- Be less verbose when Dune outputs a user message.
- Error reporting is deterministic by default.
- Prefer not to use a diff program for displaying diffs.

This list is not exhaustive and may change in the future. In order to find the exact behaviour, it is recommended to search for `INSIDE_DUNE` in the codebase.

## Inspecting Traces with jq

Dune writes trace events to `_build/trace.csexp` in canonical s-expression format. Use `dune trace cat` to read and convert these events to JSON (one object per line). The test suite includes a shared jq library at `test/blackbox-tests/dune.jq` with helper functions for common patterns.

```
# Use the shared library with include
$ dune trace cat | jq 'include "dune"; processes'

# Filter events by category
$ dune trace cat | jq 'select(.cat == "cache")'
```

Use the `-s` (slurp) flag when you need to collect all events into an array, for example when counting, sorting, or accessing first/last events:

```
$ dune trace cat | jq -s '[ .[] | select(.cat == "cache") ] | length'
$ dune trace cat | jq -s 'first | {name, cat}'
```

Enable additional trace categories with `DUNE_TRACE` (see `src/dune_trace/category.ml` for the full list):

```
$ export DUNE_TRACE=cache
$ dune build && dune trace cat | jq 'include "dune"; cacheMisses'
```

For deterministic test output, use helpers like `redactCommandTimes` to replace timing values that vary between runs:

```
$ dune trace cat | jq 'include "dune"; select(.cat == "cram") | .args |_
↪redactCommandTimes'
```

### ➔ See also

*Profiling Dune* for loading traces into `chrome://tracing`

## Guidelines

As with any long running software project, code written by one person will eventually be maintained by another. Just like normal code, it's important to document tests, especially since test suites are most often composed of many individual tests that must be understood on their own.

A well-written test case should be easily understood. A reader should be able to quickly understand what property the test is checking, how it's doing it, and how to convince oneself that the test outcome is the right one. A well-written test makes it easier for future maintainers to understand the test and react when the test breaks. Most often, the code will need to be adapted to preserve the existing behavior; however, in some rare cases, the test expectation will need to be updated.

It's crucial that each test case makes its purpose and logic crystal clear, so future maintainers know how to deal with it.

When writing a test, we generally have a good idea of what we want to test. Sometimes, we want to ensure a newly developed feature behaves as expected. Other times, we want to add a reproduction case for a bug reported by a user to ensure future changes won't reintroduce the faulty behaviour. Just like when programming, we turn such an idea into code, which is a formal language that a computer can understand. While another person reading this code might be able to follow and understand what the code does step by step, it isn't clear that they'll be able to reconstruct the original developer's idea. Even worse, they might understand the code in a completely different way, which would lead them to update it incorrectly.

### 7.3.4 Setting Up Your Development Environment Using Nix

You can use Nix to setup the development environment. This can be done by running `nix develop` in the root of the Dune repository.

Note that Dune only takes OCaml as a dependency and the rest of the dependencies are used when running the test suite.

Running `nix develop` can take a while the first time, therefore it is advisable to save the state in a profile.

```
`sh nix develop --profile nix/profiles/dune `
```

And to load the profile:

```
`sh nix develop nix/profiles/dune `
```

This profile might need to be updated from time to time, since the bootstrapped version of Dune may become stale. This can be done by running the first command.

We have the following shells for specific tasks:

- `nix develop .#slim` for a dev environment with fewer dependencies that is faster to build.
- `nix develop .#slim-melange`: same as above, but additionally includes the `melange` and `mel` packages
- Building documentation requires `nix develop .#doc`.
- For running the Rocq tests, you can use `nix develop .#rocq`. There are two classes of tests: `+ make test-rocq`: these work well on a regular Dune opam dev switch `+ make test-rocq-native`: these require the Rocq native compiler to run, and thus need OCaml 4.x

### 7.3.5 Testing Reverse Dependencies

You can test how changes to Dune affect reverse dependencies (packages that depend on Dune) using Nix. The `revdeps` flake output builds OCaml packages with a specified version of Dune.

```
# Build lwt with current version of dune
$ nix build .#revdeps.x86_64-linux.lwt --override-input revdeps-dune path:

# Build base and core with dune 3.20.2
$ nix build .#revdeps.x86_64-linux.{base,core} --override-input revdeps-dune
↳github:ocaml/dune/3.20.2

# Build all revdeps
$ nix build .#revdeps.x86_64-linux.all --override-input revdeps-dune path:
```

### Bisecting Regressions

To find which commit broke a reverse dependency, use `git bisect` with `--no-checkout` to keep the current flake in place while varying the Dune source:

```
$ git bisect start --no-checkout
$ git bisect bad HEAD
$ git bisect good 3.16.0

$ git bisect run sh -c 'nix build .#revdeps.x86_64-linux.lwt --override-input revdeps-
↳dune "github:ocaml/dune/$(git rev-parse BISECT_HEAD)''
```

The `--no-checkout` flag ensures that the working tree stays on the current branch (with the working `revdeps` output), while `BISECT_HEAD` points to the commit being tested. The Dune source is fetched from GitHub for each bisect step.

### 7.3.6 Releasing Dune

Dune's release process relies on [dune-release](#). Make sure you install and understand how this software works before proceeding. Publishing a release consists of two steps:

- Updating `CHANGES.md` to reflect the version being published.
- Running `$ make opam-release` to create the release tarball. Then publish it to GitHub and submit it to opam.

### Major & Feature Releases

Given a new version `x.y.z`, a major release increments `x`, and a feature release increments `y`. Such a release must be done from the `main` branch. Once you publish the release, be sure to publish a release branch named `x.y`.

## Point Releases

Point releases increment the z in `x.y.z`. Such releases are done from the respective `x.y` branch of the respective feature release. Once released, be sure to update `CHANGES.md` in the `main` branch.

### 7.3.7 Adding Stanzas

Adding new stanzas is the most natural way to extend Dune with new features. Therefore, we try to make this as easy as possible. The minimal amount of steps to add a new stanza is:

- Extend `Stanza.t` with a new constructor to represent the new stanza
- Modify `Dune_file` to parse the Dune language into this constructor
- Modify the rules to interpret this stanza into rules, usually done in `Gen_rules`

## Versioning

Dune is incredibly strict with versioning of new features, modifications visible to the user, and changes to existing rules. This means that any added stanza must be guarded behind the version of the Dune language in which it was introduced. For example:

```
; ( "cram"
  , let+ () = Dune_lang.Syntax.since Stanza.syntax (2, 7)
    and+ t = Cram_stanza.decode in
  [ Cram t ] )
```

Here, Dune 2.7 introduced the Cram stanza, so the user must enable `(lang dune 2.7)` in their dune project file to use it.

`since` isn't the only primitive for making sure that versions are respected. See `Dune_lang.Syntax` for other commonly used functions.

## Experimental & Independent Extensions

Sometimes, Dune's versioning policy is too strict. For example, it doesn't work in the following situations:

- When most Dune independent extensions only exist inside Dune for development convenience, e.g., build rules for Rocq. Such extensions would like to impose their own versioning policy.
- When experimental features cannot guarantee Dune's strict backwards compatibility. Such features may be dropped or modified at any time.

To handle both of these use cases, Dune allows the definition of new languages (with the same syntax). These languages have their own versioning scheme and their own stanzas (or fields). In Dune itself, `Syntax.t` represents such languages. Here's an example of how the Rocq syntax is defined:

```
let rocq_syntax =
  Dune_lang.Syntax.create ~name:"rocq" ~desc:"Rocq Prover build language"
  [ ((0, 11), `Since (3, 21)); ((0, 12), `Since (3, 22)) ]
```

The list provides which versions of the syntax are provided and which version of Dune introduced them.

Such languages must be enabled in the dune project file separately:

```
(lang dune 3.25)
(using rocq 0.13)
```

If such extensions are experimental, it's recommended that they pass `~experimental:true`, and that their versions are below 1.0.

We also recommend that such extensions introduce stanzas or fields of the form `ext_name.stanza_name` or `ext_name.field_name` to clarify which extensions provide a certain feature.

### 7.3.8 Dune Rules

#### Creating Rules

A Dune rule consists of 3 components:

- *Dependencies* that the rule may read when executed (files, aliases, etc.), described by 'a `Action_builder.t` values.
- *Targets* that the rule produces (files and/or directories), described by 'a `Action_builder.With_targets.t` values.
- *Action* that Dune must execute (external programs, redirects, etc.). Actions are represented by `Action.t` values.

Combined, one needs to produce an `Action.t Action_builder.With_targets.t` value to create a rule. The rule may then be added by `Super_context.add_rule` or a related function.

To make this maximally convenient, there's a `Command` module to make it easier to create actions that run external commands and describe their targets and dependencies simultaneously.

#### Loading Rules

Dune rules are loaded lazily to improve performance. Here's a sketch of the algorithm that tries to load the rule that generates some target file `t`.

- Get the directory that contains `t`. Call it `d`.
- Load all rules in `d` into a map from targets in that directory to rules that produce it.
- Look up the rule for `t` in this map.

To adhere to this loading scheme, we must generate our rules as part of the callback that creates targets in that directory. See the `Gen_rules` module for how this callback is constructed.

### 7.3.9 Documentation

User documentation lives in the `./doc` directory.

In order to build the user documentation, you must install `python-sphinx`, `sphinx-design`, `sphinx-copybutton`, `myst-parser`, and `furo`.

Build the documentation with

```
$ make doc
```

For automatically updated builds, you can install `sphinx-autobuild`, and run

```
$ make livedoc
```

#### See also

`doc/requirements.txt` for an always up-to-date list of packages to install

Nix users may drop into a development shell with the necessary dependencies for building docs `nix develop .#doc`.

## Structure

For structure, we use the [Diátaxis framework](#). The core idea is that documents should fit in one of the following categories:

- Tutorials, focused on learning
- How-to guides, focused on task solving
- Reference, focused on information
- Explanations, focused on understanding

Most features do not need a document in each category, but the important part is that a single document should not try to be in several categories at once.

## ReStructured Text

For code blocks containing Dune files, use `.. code:: dune` and indent with 3 spaces. Use formatting consistent with how Dune formats Dune files (most importantly, do not leave orphan closing parentheses).

In a document that only contains Dune code blocks, it is possible to use the `.. highlight:: dune` directive to have `dune` be the default lexer, and then it is possible to use the `::` shortcut to end a line with a single `:` and start a code block. See the source of [Lexical Conventions](#) for an example.

For links, prefer references that use `:doc:` (link to a whole document) or `:term:` (link to a definition in the glossary) to `:ref:`.

Use the right lexers: - `dune` for `dune` and related files - `opam` for `opam` files - `console` for shell sessions and commands (start with `$`) - `cram` for `cram` tests

## Style

Use American spelling.

Use [Title Case](#) for titles and headings (every word except “little words” like of, and, or, etc.).

For project names, use the following capitalization:

- **Dune** is the project, `dune` is the command. Files are called `dune` files.
- `dune-project` should always be written in monospace.
- **OCaml**
- **OCamlFormat**, and `ocamlformat` is the command.
- `odoc`, always in monospace.
- **opam**. Can be capitalised as `Opam` at the beginning of sentences only, as the official name is formatted `opam`. Even in titles, headers, and subheaders, it should be all lowercase: `opam`. The command is `opam`.
- **esy**. Can be capitalised as `Esy`.
- **Nix**. The command is `nix`.
- **Js\_of\_ocaml** can be abbreviated **JSOO**.
- **MDX**, rather than `mdx` or `Mdx`
- **PPX**, rather than `ppx` or `Ppx`; `ppxLib`
- **UTop**, rather than `utop` or `Utop`.

### 7.3.10 Vendoring

Dune vendors some code that it uses internally. This is done to make installing Dune easy as it requires nothing but an OCaml compiler as well as to prevent circular dependencies. Before vendoring, make sure that the license of the code allows it to be included in Dune.

The vendored code lives in the `vendor/` subdirectory. To vendor new code, create a shell script `update-<library>.sh`, that will be launched from the `vendor/` folder to download and unpack the source and copy the necessary source files into the `vendor/<library>` folder. Try to keep the amount of source code imported minimal, e.g., leave out `dune-project` files. For the most part, it should be enough to copy `.ml` and `.mli` files. Make sure to also include the license if there is such a file in the code to be vendored to stay compliant.

As these sources get vendored not as subprojects but parts of Dune, you need to deal with `public_name`. The preferred way is to remove the `public_name` and only use the private name. If that is not possible, the library can be renamed into `dune-private-libs.<library>`.

To deal with the modified dune files in `update-<library>.sh` scripts, you can commit the modified files to dune and make the `update-<library>.sh` script to use `git checkout` to restore the dune file.

For larger modifications, it is better to fork the upstream project in the [ocaml-dune](#) organisation and then vendor the forked copy in Dune. This makes the changes better visible and easier to update from upstream in the long run while keeping our custom patches in sync. The changes to the dune files are to be kept in the Dune repository.

It is preferable to cut out as many dependencies as possible, e.g., ones that are only necessary on older OCaml versions or build-time dependencies.

### 7.3.11 General Guidelines

Dune has grown to be a fairly large project that over time has acquired its own style. Below is an attempt to enumerate some important points of this style. These rules aren't axioms and we may break them when justified. However, we should have a good reason in mind when breaking them. Finally, the list isn't exhaustive by any means and is subject to change. Feel free to discuss anything in particular with the team.

- Parameter signatures should be self descriptive. Use labels when the types alone aren't sufficient to make the signature readable.

Bad:

```
val display_name : string -> string -> _ Pp.t
```

Good:

```
val display_name : first_name:string -> last_name:string -> _ Pp.t
```

- Avoid type aliases when possible. Yes, they might make some type signatures more readable, but they make the code harder to grep and make Merlin's inferred types more confusing.
- Every `.ml` file must have a corresponding `.mli`. The only exception to this rule is `.ml` files with only type definitions.
- Do not write `.mli` only modules. They offer no advantages to `.ml` modules with type definitions and one cannot define exceptions in `.mli` only modules
- Every module should have toplevel documentation that describes the module briefly. This is a good place to discuss its purpose, invariants, etc.
- Keep interfaces short & sweet. The less functions, types, etc., there are, the easier it is for users to understand, use, and ultimately modify the interface correctly. Instead of creating elaborate interfaces with the hope of future-proofing every use case, embrace change and make it easier to throw out or replace the interface.

Ideally the interface should have one obvious way to use it. A particularly annoying violator of this principle is the “logic-less chain of functions” helper. For example:

```
let foo t = bar t |> baz
```

If `bar` and `baz` are already public, then there’s no need to add yet another helper to save the caller a line of code.

- Define bindings as close to their use site as possible. When they’re far apart, reading code requires scrolling and IDE tools to understand the code.

Bad:

```
let dir = .. in
(* 50 odd lines or so that don't use [dir] *)
f dir
```

Good:

```
let dir = .. in
f dir
```

- A corollary to the previous guideline: keep the scope of bindings as small as possible.

Bad:

```
let x1 = f foo in let x2 = f bar in
let y1 = g foo in let y2 = g bar in
let dx = x2 -. x1 in
let dy = y2 -. y1 in
dx^2 +. dy^2
```

Good:

```
let dx =
  let x1 = f foo in let x2 = f bar in
  x2 -. x1
in
let dy =
  let y1 = g foo in let y2 = g bar in
  y2 -. y1
in
dx^2 +. dy^2
```

- Prefer `Code_error.raise` instead of `assert false`. The reader often has no idea what invariant is broken by the `assert false`. Kindly describe it to the reader in the error message.
- Avoid meaningless names like `x`, `a`, `b`, `f`. Try to find a more descriptive name or just inline it altogether.
- If a module `Foo` has a module type `Foo.S` and you’d like to avoid repeating its definition in the implementation and the signature, introduce an `.ml-only` module `Foo_intf` and write the `S` only once in there.
- Instead of introducing a type `foo`, consider introducing a module `Foo` with a type `t`. This is often the place to put functions related to `foo`.
- Avoid optional arguments. They increase brevity at the expense of readability and are annoying to grep. Furthermore, they encourage callers not to think at all about these optional arguments even if they often should.
- Avoid qualifying modules when accessing fields of records or constructors. Avoid it altogether if possible, or add a type annotation if necessary.

Bad:

```
let result = A.b () in
match result.A.field with
| B.Constructor -> ...
```

Good:

```
let result : A.t = A.b () in
match (result.field : B.t) with
| Constructor -> ...
```

- When constructing records, use the qualified names in in the record. Do not open the record. The local open syntax pulls in all kinds of names from the opened module and might shadow the values that you're trying to put into the record, leading to difficult debugging.

Bad; if A.value exists, it will pick that over value:

```
let value = 42 in
let record = A.{ field = value; other } in
...
```

Good:

```
let value = 42 in
let record = { A.field = value; other } in
...
```

- Stage functions explicitly with the Staged module.
- Do not raise `Invalid_argument`. Instead, raise with `Code_error.raise` which allows to attach more informative payloads than just strings.
- When ignoring the value of a let binding `let _ = ...`, we add type annotations to the ignored value `let (_ : t) = ...`. We do this convention because:
- We need to make sure we never ignore `Fiber.t` accidentally. Functions that return `Fiber.t` are always free of side effects so we need to bind on the result to force the side effect.
- Whenever a function is changed to return an error via its return value, we want the compiler to notify all the callers that need to be updated.
- To write a `to_dyn` function on a record type, use the following pattern. It ensures that the pattern matching will break when a field is added. To ignore a field, add `; d = _, not ; _`.

```
let to_dyn {a; b; c} =
  Dyn.record
  [ ("a", A.to_dyn a)
  ; ("b", B.to_dyn b)
  ; ("c", C.to_dyn c)
  ]
```

- To write an equality function, use the following pattern (this applies to other kinds of binary functions). The same remarks about about pattern matching and ignoring fields apply.

```
let equal {a; b; c} t =
  A.equal a t.a &&
```

(continues on next page)

(continued from previous page)

```
B.equal b t.b &&
C.equal c t.c
```

## Subjective Style Points

There's some stylistic decisions we made that don't have logical justification and are basically a matter of taste. Nevertheless, it's useful to follow them to keep the code consistent.

- Match patterns should be sorted by the length of their RHS when possible. Keep the shorter clauses near the top.
- If a module `Foo` defines a type `t`, all functions that take `t` in this module should have `t` as their first argument. This is the “t comes first” rule.
- Do not mix `|>` and `@@` in the same expression.
- Introduce bindings that will allow opportunities for record or label punning.
- Do not write inverted if-else expressions.

Bad:

```
(* try reading this out loud without short circuiting your brain *)
if not x then foo else bar
```

Good:

```
if x then bar else foo
```

- We prefer snake\_casing identifiers. This includes the names of modules and module types.
- Avoid qualifying constructors and record fields. Instead, add type annotations to the type being matched on or being constructed, e.g.,

Bad:

```
let foo = Command.Args.S []
```

Good:

```
let (foo : _ Command.Args.t) = S []
```

## 7.3.12 Benchmarking

### Dune Bench

You can benchmark Dune's performance by running `make bench`. This will run a subset of the Duniverse. If you are running the bench locally, make sure that you bootstrap since that is the executable that the bench will run.

The bench will build a specially selected portion of the Duniverse once, called a “clean build”. Afterwards, the build will be run 5 more times and are termed the “Null builds”.

In each run of the CI, there will be an `ocaml-benchmarks` status in the summary. Clicking `Details` will show a bench report.

The report contains the following information:

- The build times for Clean and Null builds
- The size of the `dune.exe` binary

- User CPU times for the Clean and Null builds
- System CPU times for the Clean and Null builds
- All the garbage collection stats apart from “forced collections” for Clean and Null builds

Pull requests that add new libraries are likely to increase the size of the dune binary.

Performance gains in Dune can be observed in the Clean and Null build times.

Memory usage can be observed in the garbage collection stats.

### Inline Benchmarks

Certain performance-critical parts of Dune are benchmarked using the `inline_benchmarks` library. These benchmarks are run when running the tests. Their outputs are currently not recorded and are only used to detect performance regressions.

### Build-Time Benchmarks

We benchmark the build time of Dune in every PR. The times can be found here:

<https://bench.ci.dev/ocaml/dune?worker=autumn&image=bench.Dockerfile>

### Melange Bench

We also benchmark a demo Melange project’s build time:

<https://ocaml.github.io/dune/dev/bench/>

## 7.3.13 Formatting

When changing the formatting configuration, it is possible to add the reformatting commit to the `.git-blame-ignore-revs` file. The commit will disappear from blame views. It is also possible to configure `git` to have the same behavior locally.

It is recommended to edit that file in a second PR, to make sure that the referenced commit has not changed.

#### See also

[GitHub - Ignore commits in the blame view](#)

## INDEX

### A

alias, 4

### B

build context, 4

build context root, 4

build profile, 4

build target, 4

### D

dialect, 4

### E

environment, 4

### I

installation, 4

installed world, 4

### P

package, 4

placeholder substitution, 5

project, 4

### R

root, 3

### S

scope, 4

stanza, 5

### W

workspace, 4