dune Documentation

Jérémie Dimino

Aug 20, 2019

Contents

CHAPTER 1

Quickstart

This document gives simple usage examples of dune. You can also look at examples for complete examples of projects using dune.

1.1 Building a hello world program

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

```
;; This declares the hello_world executable implemented by hello_world.ml
(executable
 (name hello_world))
```

This `hello_world.ml` file:

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

And build it with:

```
dune build hello_world.exe
```

The executable will be built as `_build/default/hello_world.exe`. Note that native code executables will have the `.exe` extension on all platforms (including non-Windows systems). The executable can be built and run in a single step with `dune exec ./hello_world.exe`.

1.2 Building a hello world program using Lwt

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`

1.3 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`

1.4 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.

1.5 Setting the OCaml compilation flags globally

Write this `dune` file at the root of your project:

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

1.6 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 are include a `config.h` file, you need to declare the dependency to this file via:

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

1.6.1 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})))
```

1.7 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)
 (c_names         mystubs)
 (c_flags         (-I/blah/include))
 (c_library_flags (-lblah)))
```

1.8 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)
 (c_names         mystubs)
 (c_flags         (:include c_flags.sexp))
 (c_library_flags (:include c_library_flags.sexp)))

(rule
 (targets c_flags.sexp c_library_flags.sexp)
 (deps    (:discover config/discover.exe))
 (action  (run %{discover} -ocamlc %{OCAMLC})))
```

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 = []
  }
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)
```

1.9 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})))
```

1.10 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.

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
```

1.11 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)
 (link_flags (-linkall))
 (modes      byte))
```

And write this in `mytoplevel.ml`

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

CHAPTER 2

Overview

Dune is a build system for OCaml and Reason. It is not intended as a completely generic build system that is able to build any given 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 hundred of developers, which means that it is highly tested and productive.

When using dune, you give very little and 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, setting up of the development tools such as merlin, etc.

In addition to the normal features one would expect from a build system for OCaml, dune provides a few additional ones that detach it from the crowd:

- you never need to tell dune where things such as libraries are. Dune will always discover them automatically. In particular, this means that when you want to re-organize your project you need to do no more than 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 always drop in the source for a dependency of your project 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 (note that dune is designed internally to make this easy but the actual support is not implemented yet)
- release directly from any revision: dune needs no setup stage. To release your project, you can simply point to a specific tag. You can of course add some release steps if you want to, but it is not necessary

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

CHAPTER 3

Terminology

- **package**: a package is a set of libraries, executables, ... that are built and installed as one by opam
- **project**: a project is a source tree, maybe containing one or more packages
- **root**: the root is the directory from where dune can build things. Dune 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**. How the root is determined is explained in *Finding the root*.
- **workspace**: the workspace is the subtree starting from the root. It can contain any number of projects that will be built simultaneously by dune
- **installed world**: anything outside of the workspace, that dune takes for granted and doesn't know how to build
- **installation**: this is the action of copying build artifacts or other files from the `<root>/_build` directory to the installed world
- **scope**: a scope determines where private items are visible. Private items include libraries or binaries that will not be installed. In dune, scopes are sub-trees rooted where at least one `<package>.opam` file is present. Moreover, scopes are exclusive. Typically, every project defines a single scope. See *Scopes* for more details
- **build context**: a build context is a subdirectory of the `<root>/_build` directory. It contains all the build artifacts of the workspace built against a specific configuration. Without specific configuration from the user, there is always a `default` build context, which corresponds to the environment in which dune is executed. Build contexts can be specified by writing a *dune-workspace* file
- **build context root**: the root of a build context named `foo` is `<root>/_build/<foo>`
- **alias: an alias is a build target that doesn't produce any file and has** configurable dependencies. Aliases are per-directory. However, on the command line, asking for an alias to be built in a given directory will trigger the construction of the alias in all children directories recursively. Dune defines several *Built-in Aliases*.
- **environment**: in dune, each directory has an environment attached to it. The environment determines the default values of various parameters, such as the compilation flags. Inside a scope, each directory inherit 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 influence various defaults. It can be set from the command line using `--profile <profile>` or from `dune-workspace` files. The following profiles are standard:
  - `release` which is the profile used for opam releases
  - `dev` which is the default profile when none is set explicitly, it has stricter warnings that the `release` one

CHAPTER 4

Project Layout and Metadata Specification

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

It is recommended to organize your project so that you 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`.

The next sections describe the format of dune metadata files.

Note that the dune metadata format is versioned in order to ensure forward compatibility. There is currently only one version available, but to be future proof, you should still specify it in your `dune` files. If no version is specified, the latest one will be used.

4.1 Metadata format

All configuration files read by Dune are using a syntax similar to the one of S-expressions, which is very simple. The Dune language can represent three kinds of values: atoms, strings and lists. By combining these, it is 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. We describe below the syntax of the language.

4.1.1 Comments

The Dune language only has end of line comments. End of line comments are introduced with a semicolon and span up to the end of the end of the current line. Everything from the semicolon to the end of the line is ignored. For instance:

```
; This is a comment
```

4.1.2 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 are always interpreted as plain backslashes characters.

4.1.3 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. Escape sequence are introduced by the a backslash character. 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, a backslash that comes just before the end of the line is used to skip the newline 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 expect a string, it will also accept an atom. As a result it possible to write most Dune configuration file using very few double quotes. This is very convenient in practice.

4.1.4 End of line strings

End of line strings are another way to write strings. The are a convenient way to write blocks of text inside a Dune file.

End of line strings are introduced by `"\|` or `"\>` and span up the end of the current line. If the next line starts as well by `"\|` or `"\>` it is the continuation of the same string. For readability, it is necessary that the text that follows the delimiter is either empty or starts with a space that is ignored.

For instance:

```
"\| this is a block
"\| of text
```

represent 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.

4.1.5 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 to represent arbitrarily complex descriptions. For instance:

```
(html
 (head (title "Hello world!"))
 (body
   This is a simple example of using S-expressions))
```

4.1.6 Variables

Dune allows variables in a few places. Their interpretation often depend on the context in which they appear.

The syntax of variables is as follow:

```
%{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.

4.2 dune-project files

These files are used to mark the root of projects as well as define project-wide parameters. These files are required to have a `lang` which controls the names and contents of all configuration files read by Dune. The `lang` stanza looks like:

```
(lang dune 1.0)
```

Additionally, they can contains the following stanzas.

4.2.1 name

Sets the name of the project. This is used by *dune subst* and error messages.

```
(name <name>)
```

4.2.2 version

Sets the version of the project:

```
(version <version>)
```

4.3 <package>.opam files

When a `<package>.opam` file is present, dune will know that the package named `<package>` exists. It will know how to construct a `<package>.install` file in the same directory to handle installation via opam. Dune also defines the recursive `install` alias, which depends on all the buildable `<package>.install` files in the workspace. So for instance to build everything that is installable in a workspace, run at the root:

```
$ dune build @install
```

Declaring a package this way will allow you to add elements such as libraries, executables, documentation, ... to your package by declaring them in `dune` files.

Such elements can only be declared in the scope defined by the corresponding `<package>.opam` file. Typically, your `<package>.opam` files should be at the root of your project, since this is where `opam pin ...` will look for them.

Note that `<package>` must be non-empty, so in particular `.opam` files are ignored.

4.3.1 Scopes

Any directory containing at least one `<package>.opam` file defines a scope. This scope is the sub-tree starting from this directory, excluding any other scopes rooted in sub-direcotries.

Typically, any given project will define a single scope. Libraries and executables that are not meant to be installed will be visible inside this scope only.

Because scopes are exclusive, if you wish to include the dependencies of the project you are currently working on into your workspace, you may copy them in a `vendor` directory, or any other name of your choice. Dune will look for them there rather than in the installed world and there will be no overlap between the various scopes.

4.3.2 Package version

Note that dune will try to determine the version number of packages defined in the workspace. While dune itself makes no use of version numbers, it can be use by external tools such as ocamlfind.

Dune determines the version of a package by trying the following methods in order:

- it looks in the `<package>.opam` file for a `version` variable
- it looks for a `<package>.version` file in the same directory and reads the first line
- it looks for the version specified in the `dune-project` if present
- it looks for a `version` file and reads the first line
- it looks for a `VERSION` file and reads the first line

`<package>.version`, `version` and `VERSION` files may be generated.

If the version can't be determined, dune just won't assign one.

4.3.3 Odig conventions

Dune follows the odig conventions and automatically installs any README*, CHANGE*, HISTORY* and LICENSE* files in the same directory as the `<package>.opam` file to a location where odig will find them.

Note that this includes files present in the source tree as well as generated files. So for instance a changelog generated by a user rule will be automatically installed as well.

4.4 jbuild-ignore (deprecated)

`jbuild-ignore` files are deprecated and replaced by *dirs (since 1.6)* stanzas in `dune` files.

CHAPTER 5

dune files

`dune` files are the main part of dune. They are used to describe libraries, executables, tests, and everything dune needs to know about.

The syntax of `dune` files is described in *Metadata format* section.

5.1 Stanzas

`dune` files are composed of stanzas. For instance a typical `dune` looks like:

```
(library
 (name mylib)
 (libraries base lwt))

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

The following sections describe the available stanzas and their meaning.

5.1.1 jbuild_version

Deprecated. This stanza is no longer used and will be removed in the future.

5.1.2 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 doesn't need to start with a uppercase letter.

For instance, the modules of a library named foo will be available as Foo.XXX outside of foo itself. It is however allowed to write an explicit Foo module, in which case this will be the interface of the library and you are free to expose only the modules you want.

Note that 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>) this is the name under which the library can be referred to as a dependency when it is not part of the current workspace, i.e. when it is installed. Without a (public_name ...) field, the library will not be installed by dune. The public name must start by the package name it is part of and optionally followed by a dot and anything else you want. The package name must be one of the packages that dune knows about, as determined by the *dune-project files*
- (synopsis <string>) should give 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 include ones that are 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 a uppercase letter. For instance to exclude module Foo: (modules (:standard \ foo))
- (libraries <library-dependencies>) is used to specify the dependencies of the library. See the section about *Library dependencies* for more details
- (wrapped <boolean>) specifies whether the modules of the library should be available only through the top-level library module, or should all be exposed at the top level. The default is true and it is 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>)) Is the same as (wrapped true) except that 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 <message> will be included in the deprecation notice for the unwrapped modules.
- (preprocess <preprocess-spec>) specifies how to preprocess files if needed. The default is no_processing. Other options are described in the *Preprocessing specification* section
- (preprocessor_deps (<deps-conf list>)) specifies extra dependencies of the preprocessor, for instance if the preprocessor reads a generated file. The specification of dependencies is described in the *Dependency specification* section
- (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. You can use this to provide extra features without adding hard dependencies to your project

- `(c_names (<names>))`, if your library has stubs, you must list the C files in this field, without the `.c` extension
- `(cxx_names (<names>))` is the same as `c_names` but for C++ stubs
- `(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
- `(modes <modes>)` 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 is not available
- `(no_dynlink)` is to disable dynamic linking of the library. This is for advanced use only, by default you shouldn't set this option
- `(kind <kind>)` is the kind of the library. The default is `normal`, other available choices are `ppx_rewriter` and `ppx_deriver` and must be set when the library is intended to be used as a ppx rewriter or a `[@@deriving ...]` plugin. The reason why `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>` is a string that supports *Variables expansion* evaluated by each invocation of the preprocessor (note: libraries that share cookies with the same name should agree on their expanded value)
- `(ppx_runtime_libraries (<library-names>))` 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. This is for instance the case of `ctypes` which will only install `ctypes.foreign` if the dummy `ctypes-foreign` package is installed. You can specify such virtual dependencies here. You don't need to do so unless you use dune to synthesize the `depends` and `depopts` sections of your opam file
- `js_of_ocaml`. See the section about *js_of_ocaml*
- `flags`, `ocamlc_flags` and `ocamlopt_flags`. See the section about *OCaml flags*
- `(library_flags (<flags>))` is a list of flags that are passed as it 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 expansion*
- `(c_flags <flags>)` specifies the compilation flags for C stubs, using the *Ordered set language*. This field supports `(:include ...)` forms
- `(cxx_flags <flags>)` is the same as `c_flags` but for C++ stubs
- `(c_library_flags <flags>)` specifies the flags to pass 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 are writing bindings for a C library named `bar`, you should typically write `-lbar` here, or whatever flags are necessary to to link against this library
- `(self_build_stubs_archive <c-libname>)` indicates to dune that the library has stubs, but that the stubs are built manually. The aim of the field is to embed a library written in foreign language and/or building with another build system. It is not for casual uses, see the re2 library for an example of use
- `(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 is not reasonably possible for dune to check that this is the case, dune requires the user to explicitly list such modules to avoid surprises. `<modules>` must be a subset of the modules listed in the `(modules ...)` field.

- `(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.
- `(allow_overlapping_dependencies)` allows external dependencies to overlap with libraries that are present in the workspace
- `(no_keep_locs)` does nothing. It used to be a necessary hack when we were waiting for proper support for virtual libraries. Do not use in new code, it will be deleted in dune 2.0
- `(enabled_if <blang expression>)` allows to conditionally disable a library. A disabled library cannot be built and will not be installed. The condition is specified using the *blang*, 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`

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 ...))`.

5.1.3 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 main entry point of the executable. 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`, `<name>.bc` and `<name>.bc.js`. `<name>.exe` is a native code executable, `<name>.bc` is a bytecode executable which requires `ocamlrun` to run and `<name>.bc.js` is a JavaScript generated using js_of_ocaml.

Note that in case native compilation is not available, `<name>.exe` will in fact be a custom byte-code executable. Custom in the sense of `ocamlc -custom`, meaning that it is a native executable that embeds the `ocamlrun` virtual machine as well as the byte code. As such 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. This is because running a byte-code executable often requires loading shared libraries that are locally built, and so requires additional setup such as setting specific environment variables and dune doesn't do at the moment.

Native compilation is considered not available when there is no `ocamlopt` binary at the same place as where `ocamlc` was found.

Executables can also be linked as object or shared object files. See *linking modes* for more information.

`<optional-fields>` are:

- `(public_name <public-name>)` specifies that the executable should be installed under that name. It is the same as adding the following stanza to your `dune` file:

  ```
  (install
   (section bin)
   (files (<name>.exe as <public-name>)))
  ```

- `(package <package>)` if there is a `(public_name ...)` field, this specifies the package the executables are part of
- `(libraries <library-dependencies>)` specifies the library dependencies. See the section about *Library dependencies* for more details

- `(link_flags <flags>)` specifies additional flags to pass to the linker. This field supports `(:include ...)` forms
- `(link_deps (<deps-conf list>))` specifies the dependencies used only by the linker, for example when using a version script. See the *Dependency specification* section 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*
- `(modes (<modes>))` sets the *linking modes*. The default is `(byte exe)`
- `(preprocess <preprocess-spec>)` is the same as the `(preprocess ...)` 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*
- **`flags`, `ocamlc_flags` and `ocamlopt_flags`. See the section about** specifying *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*
- `(promote <options>)` allows to promote 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 `dune clean` to delete them

Linking modes

The `modes` field allows to select what linking modes should be used to link executables. Each mode is a pair `(<compilation-mode> <binary-kind>)` where `<compilation-mode>` describes whether the byte code or native code backend of the OCaml compiler should be used and `<binary-kind>` describes what kind of file should be produced.

`<compilation-mode>` must be `byte`, `native` or `best`, where `best` is `native` with a fallback to byte-code when native compilation is not 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*.

For instance the following `executables` stanza will produce byte code executables and native shared objects:

```
(executables
 ((names (a b c))
  (modes ((byte exe) (native shared_object)))))
```

Additionally, you can use the following short-hands:

- `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)`

For instance the following `modes` fields are all equivalent:

```
(modes (exe object shared_object))
(modes ((best exe)
        (best object)
        (best shared_object)))
```

The extensions for the various linking modes are chosen as follows:

compilation mode	binary kind	extensions
byte	exe	.bc and .bc.js
native/best	exe	.exe
byte	object	.bc%{ext_obj}
native/best	object	.exe%{ext_obj}
byte	shared_object	.bc%{ext_dll}
native/best	shared_object	%{ext_dll}
byte	c	.bc.c
byte	js	.bc.js

Where `%{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`.

Note that when `(byte exe)` is specified but neither `(best exe)` nor `(native exe)` are specified, Dune still knows how to build an executable with the extension `.exe`. In such case, the `.exe` version is the same as the `.bc` one except that it is linked with the `-custom` option of the compiler. You should always use the `.exe` rather that the `.bc` inside build rules.

5.1.4 executables

The `executables` stanza is the same as the `executable` stanza, except that it is used to describe several executables sharing the same configuration.

It shares the same fields as the `executable` stanza, except that instead of `(name ...)` and `(public_name ...)` you must use:

- `(names <names>)` where `<names>` is a list of entry point names. As for `executable` you only need to specify the modules containing the entry point of each executable
- `(public_names <names>)` describes under what name each executable should be installed. 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

5.1.5 rule

The `rule` stanza is used to create custom user rules. It tells dune how to generate a specific set of files from a specific set of dependencies.

The syntax is as follows:

```
(rule
 (target[s] <filenames>)
 (action  <action>)
 <optional-fields>)
```

`<filenames>` is a list of file names (if defined with `targets`) or exactly one file name (if defined with `target`). Note that currently dune only supports user rules with targets in the current directory.

`<action>` is the action to run to produce the targets from the dependencies. See the *User actions* section for more details.

`<optional-fields>` are:

- `(deps <deps-conf list>)` to specify the dependencies of the rule. See the *Dependency specification* section for more details.
- `(mode <mode>)` to specify how to handle the targets, see *modes* for details
- `(fallback)` is deprecated and is the same as `(mode fallback)`
- `(locks (<lock-names>))` specify that the action must be run while holding the following locks. See the *Locks* section for more details.

Note that 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.

modes

By default, the target of a rule must not exist in the source tree and dune will error out when this is the case.

However, it is possible to change this behavior using the `mode` field. The following modes are available:

- `standard`, this is the standard mode
- `fallback`, in this mode, when the targets are already present in the source tree, dune will ignore the rule. It is an error if only a subset of the targets are present in the tree. The common use of fallback rules is 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 is available since Dune 1.8 - `(only <predicate>)` means that only a subset of the targets should be promoted. The argument is a predicate in a syntax similar to the argument of *(dirs ...)*. This feature is available since dune 1.10
- `promote-until-clean` is the same as `(promote (until-clean))`
- `(promote-into <dir>)` is the same as `(promote (into <dir>))`
- `(promote-until-clean-into <dir>)` is the same as `(promote (until-clean) (into <dir>))`

The `(promote <options>)` form is only available since Dune 1.10. Before Dune 1.10, you need to use one of the `promote-...` forms. The `promote-...` forms should disappear in Dune 2.0, so using the more generic `(promote <options>)` form should be prefered in new projects.

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 will `(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 promotes only a subset of their targets via `(only ...)` are never ignored.

inferred rules

When using the action DSL (see *User actions*), it is most of the time obvious what are the dependencies and targets.

For instance:

```
(rule
 (target b)
 (deps   a)
 (action (copy %{deps} %{target})))
```

In this example it is obvious by inspecting the action what the dependencies and targets are. When this is the case you can use the following shorter syntax, where dune infers dependencies and targets for you:

```
(rule <action>)
```

For instance:

```
(rule (copy a b))
```

Note that in dune, targets must always be known statically. Especially, this mean that dune must be able to statically determine all targets. For instance, this `(rule ...)` stanza is rejected by dune:

```
(rule (copy a b.%{read:file}))
```

5.1.6 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>))
```

5.1.7 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>))
```

5.1.8 menhir

A `menhir` stanza is available to support the *menhir* parser generator. See the *Menhir* section for details.

5.1.9 cinaps

A `cianps` stanza is available to support the `cinaps` tool. See the cinaps website for more details.

5.1.10 alias

The `alias` stanza lets you add dependencies to an alias, or specify an action to run to construct the alias.

The syntax is as follows:

```
(alias
 (name    <alias-name>)
 (deps    <deps-conf list>)
 <optional-fields>)
```

`<name>` is an alias name such as `runtest`.

`<deps-conf list>` specifies the dependencies of the alias. See the *Dependency specification* section for more details.

`<optional-fields>` are:

- `<action>`, an action to run when constructing the alias. See the *User actions* section for more details.
- `(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>))` specify that the action must be run while holding the following locks. See the *Locks* section for more details.
- `(enabled_if <blang expression>)` specifies the boolean condition that must be true for the tests to run. The condition is specified using the *blang*, and the field allows for *variables* to appear in the expressions.

The typical use of the `alias` stanza is to define tests:

```
(alias
 (name   runtest)
 (action (run %{exe:my-test-program.exe} blah)))
```

See the section about *Running tests* for details.

Note that if your project contains several packages and you run the tests from the opam file using a `build-test` field, then all your `runtest` alias stanzas should have a `(package ...)` field in order to partition the set of tests.

5.1.11 install

Dune supports installing packages on the system, i.e. copying freshly built artifacts from the workspace to the system. See the *installation* section for more details.

Handling of the .exe extension on Windows

Under Microsoft Windows, executables must be suffixed with `.exe`. Dune tries to make sure that executables are always installed with this extension on Windows.

More precisely, when installing a file via an `(install ...)` stanza, if the source file has extension `.exe` or `.bc`, then dune implicitly adds the `.exe` extension to the destination, if not already present.

5.1.12 copy_files

The `copy_files` and `copy_files#` stanzas allow to specify that files from another directory could be copied if needed to the current directory.

The syntax is as follows:

```
(copy_files <glob>)
```

`<glob>` represents the set of files to copy, see the *glob* for details.

The difference between `copy_files` and `copy_files#` is the same as the difference between the `copy` and `copy#` action. See the *User actions* section for more details.

5.1.13 include

The `include` stanza allows to include the contents of another file into the current dune file. Currently, the included file cannot be generated and must be present in the source tree. This feature is intended to be used 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)))

(alias
 (name   runtest)
 (action (diff dune.inc dune.inc.gen)))
```

With this dune file, running dune as follow will replace the `dune.inc` file in the source tree by the generated one:

```
$ dune build @runtest --auto-promote
```

5.1.14 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 will define an executable named `mytest.exe` that will be executed as part of 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 that are 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.

By default the test binaries are run without options. The `action` field can be used to override the test binary invocation, for example 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)))
```

5.1.15 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>)
```

where the `name` field is singular. The same optional fields are supported.

5.1.16 env

The `env` stanza allows to modify the environment. The syntax is as follow:

```
(env
 (<profile1> <settings1>)
 (<profile2> <settings2>)
 ...
 (<profilen> <settingsn>))
```

The first form `(<profile> <settings>)` that correspond to the selected build profile will be used to modify the environment in this directory. You can use _ to match any build profile.

Fields supported in `<settings>` are:

- any OCaml flags field, see *OCaml flags* for more details.
- `(c_flags <flags>)` and `(cxx_flags <flags>)` to specify compilation flags for C and C++ stubs, respectively. See *library* for more details.
- `(env-vars (<var1> <val1>) .. (<varN> <valN>))`. This will add the corresponding variables to the environment in which the build commands are executed, and under which `dune exec` runs. At the moment, this mechanism is only supported in `dune-workspace` files.

- `(binaries <filepath> (<filepath> as <name>))`. This will make the binary at `<filepath>` as `<name>`. If the `<name>` isn't provided, then it will be inferred from the basename of `<filepath>` by dropping the `.exe` suffix if it exists.
- `(inline_tests <state>)` where state is either `enabled`, `disabled` or `ignored`. This field is available since Dune 1.11. It controls the value of the variable `%{inline_tests}` that is read by the inline test framework. The default value is `disabled` for the `release` profile and `enabled` otherwise.

5.1.17 dirs (since 1.6)

The `dirs` stanza allows to tell specify the sub-directories dune will include in a build. The syntax is based on dune's predicate language and allows the user 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 directories except ocaml
(dirs :standard \ test* foo*) ;; exclude all directories that start with test or foo
```

A directory that is not included by this stanza will not be eagerly scanned by Dune. Any `dune` or other special files in it won't be interpreted either and will be treated as raw data. It is however possible to depend on files inside ignored sub-directories.

5.1.18 data_only_dirs (since 1.6)

Dune allows the user to treat directories as *data only*. Dune files in these directories will not be evaluated for their rules, but the contents of these directories will still be usable as dependencies for other rules.

The syntax is the same as for the `dirs` stanza except that `:standard` is by default empty.

Example:

```
;; dune files in fixtures_* dirs are ignored
(data_only_dirs fixtures_*)
```

5.1.19 ignored_subdirs (deprecated in 1.6)

One may also specify *data only* directories using the `ignored_subdirs` stanza. The meaning is the same as `data_only_dirs` but the syntax isn't as flexible and only accepts a list of directory names. It is 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> ...)
```

5.1.20 vendored_dirs (since 1.11)

Dune supports vendoring of 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 workaround those by explicitly marking such directories as containing vendored code.

Example:

```
(vendored_dirs vendor)
```

Dune will not resolve aliases in vendored directories meaning by default it will not build all installable targets, run the test, format or lint the code located in such a directory while still building the parts your project depend upon. Libraries and executable in vendored directories will also be built with a `-w -a` flag to suppress all warnings and prevent pollution of your build output.

5.1.21 include_subdirs

The `include_subdirs` stanza is used to control how dune considers sub-directories of the current directory. The syntax is as follow:

```
(include_subdirs <mode>)
```

Where `<mode>` maybe be one of:

- `no`, the default
- `unqualified`

When the `include_subdirs` stanza is not present or `<mode>` is `no`, dune considers sub-directories as independent. When `<mode>` is `unqualified`, dune will assume that the sub-directories of the current directory are part of the same group of directories. In particular, dune will scan all these directories at once when looking for OCaml/Reason files. This allows you to split a library between several directories. `unqualified` means that modules in sub-directories 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. It is planned to add a `qualified` mode in the future.

Note that sub-directories are included recursively, however the recursion will stop when encountering a sub-directory that contains another `include_subdirs` stanza. Additionally, it is not allowed for a sub-directory of a directory with `(include_subdirs <x>)` where `<x>` is not `no` to contain one of the following stanzas:

- `library`
- `executable(s)`
- `test(s)`

5.1.22 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
```

5.1.23 external_variant

The `external_variant` allow to declare a tagged implementation that does not live inside the virtual library project.

```
(external_variant
 (variant foo)
 (implementation lib-foo)
 (virtual_library vlib))
```

This will add *lib-foo* to the list of known implementations of *vlib*. For more details see *Variants*

5.2 Common items

5.2.1 Ordered set language

A few fields takes as argument an ordered set and can be specified using a small DSL.

This DSL is interpreted by dune into an ordered set of strings using the following rules:

- `:standard` denotes the standard value of the field when it is 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>)`. This is useful for instance 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, the first element of a list cannot be an atom except if it starts with - or *:*. The reason for this is that we are planning to add simple programmatic features in the futures so that one may write:

```
(flags (if (>= %{ocaml_version} 4.06) ...))
```

This restriction will allow to add this feature without introducing a breaking changes. If you want to write a list where the first element doesn't start by -, you can simply quote it: `("x" y z)`.

Most fields using the ordered set language also support *Variables expansion*. Variables are expanded after the set language is interpreted.

5.2.2 Boolean Language

The boolean language allows the user to define simple boolean expressions that dune can evaluate. Here's a semi formal specification of the language:

```
op := '=' | '<' | '>' | '<>' | '>=' | '<='

expr := (and <expr>+)
      | (or <expr>+)
      | (<op> <template> <template>)
      | <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.

Here's a simple example of a condition that expresses running on OSX and having an flambda compiler with the help of variable expansion:

```
(and %{ocamlc-config:flambda} (= %{ocamlc-config:system} macosx))
```

5.2.3 Variables expansion

Some fields can contains variables of the form `%{var}` that are expanded by dune.

Dune supports the following variables:

- `project_root` is the root of the current project. It is typically the toplevel directory of your project 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. Note that the value of `workspace_root` is not 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 was used to compile OCaml in the current build context
- `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 targeting 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 extension used for various artifacts
- `ocaml-config:v` for every variable `v` in the output of `ocamlc -config`. 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 versions of OCaml needs to be registered in dune before they can be used
- `profile` the profile selected via `--profile`
- `context_name` the name of the context (`default` or defined in the workspace file)
- `os_type` is the type of the OS the build is targetting. This is the same as `ocaml-config:os_type`
- `architecture` is the type of the architecture the build is targetting. This is the same as `ocaml-config:architecture`

- `model` is the type of the cpu the build is targetting. This is the same as `ocaml-config:model`
- `system` is the name of the OS the build is targetting. This is the same as `ocaml-config:system`
- `ignoring_promoted_rule` is `true` if `--ignore-promoted-rules` was passed on the command line and `false` otherwise

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>` is the same as `<path>`, except when cross-compiling, in which case it will expand to `<path>` from the host build context
- `bin:<program>` expands to a path to `program`. If `program` is installed by a package in the workspace (see *install* stanzas), the locally built binary will be used, otherwise it will be searched in the `PATH` of the current build context. Note that `(run %{bin:program} ...)` and `(run program ...)` behave in the same way. `%{bin:...}` is only necessary when you are using `(bash ...)` or `(system ...)`
- `lib:<public-library-name>:<file>` expands to a path to file `<file>` of 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
- `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
- `lib-available:<library-name>` expands to `true` or `false` depending on whether the library is available or not. A library is available iff at least one of the following condition holds:
  - it is part the installed worlds
  - it is available locally and is not optional
  - it is available locally and all its library dependencies are available
- `version:<package>` expands to the version of the given package. Note that this is only supported for packages that are being defined in the current scope
- `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

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
```

1. Expansion of lists

Forms that expands to list of items, such as `%{cc}`, `%{deps}`, `%{targets}` or `%{read-lines:...}`, are suitable to be used in, say, `(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 the two dependencies to be passed as a single argument, you have to quote the variable as in:

```
(run foo "%{deps}")
```

which is equivalent to the following shell command:

```
$ foo "a b"
```

(the items of the list are concatenated with space). Note that, 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)
 (deps   foo.c)
 (action (run %{cc} -o %{target} %{deps} -lfoolib)))
```

5.2.4 Library dependencies

Dependencies on libraries are specified using `(libraries ...)` fields in `library` and `executables` stanzas.

For libraries defined in the current scope, you can use either 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, libraries that are part of the workspace are always preferred to ones that are part of the installed world.

Alternative dependencies

In addition to direct dependencies you can specify alternative dependencies. This is described in the *Alternative dependencies* section

It is sometimes the case that one wants to not depend on a specific library, but instead on whatever is already installed. For instance 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.

5.2.5 Preprocessing specification

Dune accepts three kinds of preprocessing:

- `no_preprocessing`, meaning that files are given as it to the compiler, this is the default
- `(action <action>)` to preprocess files using the given action
- `(pps <ppx-rewriters-and-flags>)` to preprocess files using the given list of ppx rewriters
- `(staged_pps <ppx-rewriters-and-flags>)` is similar to `(pps ...)` but behave slightly differently and is needed for certain ppx rewriters (see below for details)
- `future_syntax` is a special value that brings some of the newer OCaml syntaxes to older compilers. See *Future syntax* for more details

Dune normally assumes that the compilation pipeline is sequenced as follow:

- code generation (including preprocessing)
- dependency analysis
- compilation

Dune uses this fact to optimize the pipeline and in particular share the result of code generation and preprocessing between the dependency analysis and compilation phases. However, some specific code generators or preprocessors require feedback from the compilation phase. As a result they must be applied in stages as follows:

- first stage of code generation
- dependency analysis
- second step of code generation in parallel with compilation

This is the case for ppx rewriters using the OCaml typer for instance. When using such ppx rewriters, you must use `staged_pps` instead of `pps` in order to force Dune to use the second pipeline, which is slower but necessary in this case.

Preprocessing with actions

`<action>` uses the same DSL as described in the *User actions* section, and for the same reason given in that section, it will be executed from the root of the current build context. It is expected to be an action that reads the file given as only dependency named `input-file` and outputs the preprocessed file on its standard output.

More precisely, `(preprocess (action <action>))` acts as if you had setup 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}")`.

Preprocessing with ppx rewriters

`<ppx-rewriters-and-flags>` is expected to be a sequence where each element is either a command line flag if starting with a `-` or the name of a library. If you want to pass command line flags that do not 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`.

Libraries listed here should be libraries implementing an OCaml AST rewriter and registering themselves using the ocaml-migrate-parsetree.driver API.

Dune will build a single executable by linking all these libraries and their dependencies. Note that it is important that all these libraries are linked with `-linkall`. Dune automatically uses `-linkall` when the `(kind ...)` field is set to `ppx_rewriter` or `ppx_deriver`.

Per module preprocessing specification

By default a preprocessing specification will apply to all modules in the library/set of executables. It is possible to select the preprocessing on a module-by-module basis by using the following syntax:

```
(preprocess (per_module
             (<spec1> <module-list1>)
             (<spec2> <module-list2>)
             ...))
```

Where `<spec1>`, `<spec2>`, ... are preprocessing specifications and `<module-list1>`, `<module-list2>`, ... are list of module names.

For instance:

```
(preprocess (per_module
             (((action (run ./pp.sh X=1 %{input-file})) foo bar))
             (((action (run ./pp.sh X=2 %{input-file})) baz))))
```

Future syntax

The `future_syntax` preprocessing specification 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.

5.2.6 Dependency specification

Dependencies in `dune` files can be specified using one of the following syntax:

- `(:name <dependencies>)` will bind the 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>)`: depend on the construction of this alias, for instance: `(alias src/runtest)`
- `(alias_rec <alias-name>)`: depend 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)`,...
- `(glob_files <glob>)`: depend on all files matched by `<glob>`, see the *glob* for details
- `(source_tree <dir>)`: depend on all source files in the subtree with root `<dir>`
- `(universe)`: depend 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>)` depend on all files installed by `<package>`, as well as on the transitive package dependencies of `<package>`. This can be used to test a command against the files that will be installed
- `(env_var <var>)`: depend on the value of the environment variable `<var>`. If this variable becomes set, becomes unset, or changes value, the target will be rebuilt.

In all these cases, the argument supports *Variables expansion*.

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 such named dependency list can also include unnamed dependencies (like `index.html` in the example above). Also, such user defined names will shadow built in variables. So `(:workspace_root x)` will shadow the built in `%{workspace_root}` variable.

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`.

Currently dune only support globbing files in a single directory. And in particular 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

The glob syntax is interpreted as follows:

- `\<char>` matches exactly `<char>`, even if it is 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>`, ...

5.2.7 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 default value for `(flags ...)` is taken from the environment, as a result it is recommended to write `(flags ...)` fields as follows:

```
(flags (:standard <my options>))
```

5.2.8 js_of_ocaml

A *js_of_ocaml field* exists in executable and libraries stanzas that allows one to customize options relevant to jsoo.

5.2.9 User 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 is interpreted by dune directly and doesn't require an external shell. All atoms in the DSL support *Variables expansion*. 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 is 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.

The following constructions are available:

- `(run <prog> <args>)` to execute a program. `<prog>` is resolved locally if it is available in the current workspace, otherwise it is resolved using the `PATH`
- `(chdir <dir> <DSL>)` to change the current directory
- `(setenv <var> <value> <DSL>)` to set an environment variable
- `(with-<outputs>-to <file> <DSL>)` to redirect the output to a file, where `<outputs>` is one of: `stdout`, `stderr` or `outputs` (for both `stdout` and `stderr`)
- `(ignore-<outputs> <DSL)` to ignore the output, where `<outputs>` is one of: `stdout`, `stderr` or `outputs`
- `(progn <DSL>...)` to execute several commands in sequence
- `(echo <string>)` to output a string on stdout
- `(write-file <file> <string>)` writes `<string>` to `<file>`
- `(cat <file>)` to print the contents of a file to stdout
- `(copy <src> <dst>)` to copy a file
- `(copy# <src> <dst>)` to copy a file and add a line directive at the beginning
- `(system <cmd>)` to execute a command using the system shell: `sh` on Unix and `cmd` on Windows
- `(bash <cmd>)` to execute a command using `/bin/bash`. This is obviously not very portable
- `(diff <file1> <file2>)` is similar to `(run diff <file1> <file2>)` but is better and allows promotion. See *Diffing and promotion* for more details
- `(diff? <file1> <file2>)` is the same as `(diff <file1> <file2>)` except that it is ignored when `<file1>` or `<file2>` doesn't exists
- `(cmp <file1> <file2>)` is similar to `(run cmp <file1> <file2>)` but allows promotion. See *Diffing and promotion* for more details

As mentioned `copy#` inserts a line directive at the beginning of the destination file. More precisely, it inserts the following line:

```
# 1 "<source file name>"
```

Most languages recognize such lines and update their current location, in order to report errors in the original file rather than the copy. This is important as the copy exists only under the `_build` directory and in order for editors to jump to errors when parsing the output of the build system, 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.

Note: expansion of the special `%{<kind>:...}` is done relative to the current working directory of the part of the DSL being executed. So for instance if you have this action in a `src/foo/dune`:

```
(action (chdir ../../.. (echo %{path:dune})))
```

Then `%{path: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 you editor might think that `blah.ml` is at the root of your project. What you should write instead is:

```
(rule
 (target blah.ml)
 (deps   blah.mll)
 (action (chdir %{workspace_root} (run ocamllex -o %{target} %{deps}))))
```

5.2.10 Locks

Given two rules that are independent, dune will assume that there associated action can be run concurrently. Two rules are considered independent if none of them depend on the other, either directly or through a chain of dependencies. This basic assumption allows 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:

```
(alias
 (name   runtest)
 (deps   foo)
 (locks  m)
 (action (run test.exe %{deps})))

(alias
 (name   runtest)
 (deps   bar)
 (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:

```
(alias
 (name   runtest)
```

(continues on next page)

(continued from previous page)

```
 (deps   foo)
 (locks  /tcp-port/1042)
 (action (run test.exe %{deps})))
```

5.2.11 Diffing and promotion

`(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 doesn't 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 to diff files can be configured via the `--diff-command` command line argument. Note that it is 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 the end of lines before comparing the files
- since `(diff a b)` is a builtin action, dune knowns that `a` and `b` are needed and 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
- it allows promotion. See below

Note that `(cmp a b)` does no end of lines normalization and doesn't print a diff when the files differ. `cmp` is meant to be used with binary files.

Promotion

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)))

(alias
 (name   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 is what you expect
- 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.

5.3 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 is executed in.

The OCaml syntax gives you an escape hatch for when the S-expression syntax is not enough. It is not 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.

CHAPTER 6

Executables

This section describes how to build and install binary programs with Dune.

6.1 Definig executables

TOOD.

6.2 Embedding build information into executables

Dune allows to embed build information such as versions in executbles via the special `dune.build-info` library. This library exposes a few informations 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)
```

For libraries and executables from development repositories that don't have version informations 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
```

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 ontained from the the version control system, the version string will only be written in the binary once it is installed or promoted to the source tree. In particular, if you evalute this expression as part of the build of your package, it will return `None`. This is to ensure that committing does not hurt your development experience. Indeed, if dune stored the version directly inside the freshly built binaries,

then everytime you commit your code the version would change and dune would need to rebuild all the binaries and everything that depend on them, such as tests. Instead Dune leaves a placeholder inside the binary and fills it during installation or promotion.

CHAPTER 7

Writing and running tests

Dune tries to streamline the testing story as much as possible, so that you can focus on the tests themselves and not bother with setting up with various test frameworks.

In this section, we will explain the workflow to deal with tests in dune. In particular we will see how to run the testsuite of a project, how to describe your tests to dune and how to promote tests result as expectation.

We distinguish two kinds of tests: inline tests and custom tests. Inline tests are usually written directly inside the ml files of a library. They are the easiest to work with and usually requires nothing more than writing `(inline_tests)` inside your library stanza. Custom tests consist on executing an executable and sometimes do something afterwards, such as diffing its output.

7.1 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. This will run all the tests defined in the current directory and any sub-directory recursively.

Note that in any case, `dune runtest` is simply a short-hand for building the `runtest` alias, so you can always ask dune to run the tests in conjunction with other targets by passing `@runtest` to `dune build`. For instance:

```
$ dune build @install @runtest
$ dune build @install @test/runtest
```

7.1.1 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
```

7.1.2 Running tests in a directory

You can also pass a directory argument to run the tests from a sub-tree. For instance `dune runtest test` will only run the tests from the `test` directory and any sub-directory of `test` recursively.

7.2 Inline tests

There are several inline tests framework available for OCaml, such as ppx_inline_test and qtest. We will use ppx_inline_test as an example as at the time of writing this document it has the necessary setup to be used with dune out of the box.

ppx_inline_test allows to write tests directly inside ml files as follow:

```
let rec fact n = if n = 1 then 1 else n * fact (n - 1)

let%test _ = fact 5 = 120
```

The file has to 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 instruct dune that our library contains inline tests, all we have to do is 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
```

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 and it is 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 since it cannot guess it. 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.

7.2.1 Inline expectation tests

Inline expectation tests are a special case of inline tests where you write a bit of OCaml code that 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 follow:

```
(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 is possible to make the editor do the promotion and make the workflow even smoother.

7.2.2 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 sub-directories.

7.2.3 Running tests in bytecode or javascript

By default Dune run inline tests in native mode, except if native compilation is not available in which case it runs them in bytecode.

You can change this setting to choose which modes tests should run in. To do that, 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 nodejs

For instance:

```
(library
 (name foo)
 (inline_tests (modes byte best js))
 (preprocess (pps ppx_expect)))
```

7.2.4 Specifying inline test dependencies

If your tests are reading files, you must say it to dune by adding a `deps` field the 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)))
```

7.2.5 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*.

7.2.6 Using additional libraries in the test runner

When tests are not part of the library code, it is 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)))
```

7.2.7 Defining your own inline test backend

If you are writing a test framework, or for 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 the user must use when they want to write tests, then you should define this library has a backend. Otherwise simply create an empty library with the name you want to give for your backend.

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 thee parameters:

1. How to create the test runner
2. How to build the test runner
3. How to run the test runner

These three parameters can be specified inside the `inline_tests.backend` field, which accepts the following fields:

```
(generate_runner   <action>)
(runner_libraries (<ocaml-libraries>))
(flags             <flags>)
(extends          (<backends>))
```

For instance:

`<action>` follows the *User actions* specification. It describe an action that should be executed in the directory of libraries using this backend for their tests. It is expected that the action produces some OCaml code on its standard output. This code will constitute the test runner. The action can use the following additional variables:

- `%{library-name}` which is the name of the library being tested
- `%{impl-files}` which is the list of implementation files in the library, i.e. all the `.ml` and `.re` files
- `%{intf-files}` which is 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 ()`, the you should probably put `my_test_framework` in the `runner_libraries` field.

If you test runner needs specific flags, you should pass them in the `flags` field. You can use the `%{library-name}` variable in this field.

Finally, a backend can be an extension of another backend. In this case you must specify by in the `extends` field. For instance, ppx_expect is an extension of ppx_inline_test. It is 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 is not 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 of 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 do not recommend using `sed` in your build as this would cause portability problems.

7.3 Custom tests

We said in *Running tests* that to run tests dune simply builds the `runtest` alias. As a result, to define custom tests, you simply need to add an action to this alias in any directory. For instance if you have a binary `tests.exe` that you want to run as part of running your testsuite, simply add this to a dune file:

```
(alias
 (name   runtest)
 (action (run ./tests.exe)))
```

Hence to define an a test a pair of alias and executable stanzas are required. To simplify this common pattern, dune provides a *tests* stanza to define multiple tests and their aliases at once:

```
(tests (names test1 test2))
```

7.3.1 Diffing the result

It is often the case that we want to compare the output of a test to some expected one. For that, dune offers the `diff` command, which in essence is the same as running the `diff` tool, except that it is more integrated in dune and especially with the `promote` command. For instance let's consider this test:

```
(rule
 (with-stdout-to tests.output (run ./tests.exe)))

(alias
 (name runtest)
 (action (diff tests.expected test.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))
```

Where dune expects a `tests.expected` file to exist to infer that this is an expect tests.

This provides a nice way of dealing with the usual *write code*, *run*, *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 that the standard `diff` tool. You can change that by passing `--diff-command CMD` to dune.

CHAPTER 8

Dealing with foreign libraries

The OCaml programming language allows to interface 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, 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

Note that 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 is too limited for building complex libraries written in C or other languages. For such cases, Dune allows to integrate a foreign build system into a normal Dune build.

8.1 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 via the `c_names` field of the *library* stanza. For instance:

```
(library
 (name mylib)
 (c_names file1 file2))
```

Similarly, you can add C++ stubs to an OCaml library by listing them without the `.cpp` extension via the `cxx_names` field.

Dune is currently not flexible regarding the extension of the C/C++ source files. They have to be `.c` and `.cpp`. If you have source files that that do not follow this extension and you want to build them with Dune, you need to rename them first. Alternatively, you can use the *foreign build sandboxing* method described bellow.

8.1.1 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.

8.1.2 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 install header files, these are made visible to users of the library via the include search path.

8.2 Foreign build sandboxing

When the build of a C library is too complicated to express in the Dune language, it is 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 *ignored_subdirs* stanza
- write a custom rule that:
  - depend on this directory recursively via *source_tree*
  - invoke the external build system
  - copy the C archive files (`.a`, `.so`, ...) in main library directory with a specific names (see bellow)
- *attach* the C archive files to an OCaml library via the *self_build_stubs_archive* field

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`:

```
(ignored_subdirs (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_stubs.a dllfoo_stubs.so)
 (action (progn
          (chdir libfoo (run make)))
          (copy libfoo/libfoo.a libfoo_stubs.a)
          (copy libfoo/libfoo.so dllfoo_stubs.so)))
```

Note that the rule copies the files to `libfoo_stubs.a` and `dllfoo_stubs.so`. It is important that the files produced are named `lib<ocaml-lib-name>_stubs.a` and `dll<ocaml-lib-name>_stubs.so`.

The last step is to attach these archives to an OCaml library as follows:

```
(library
 (name bar)
 (self_build_stubs_archive foo))
```

Then, whenever you use the `bar` library, you will also be able to use C functions from `libfoo`.

8.2.1 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 are interested in helping make this happen, please let the Dune team know and someone will guide you.

8.2.2 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.

CHAPTER 9

Generating Documentation

9.1 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 likely be done with opam:

```
$ opam install odoc
```

9.2 Writing Documentation

Documentation comments will be automatically extracted from your OCaml source files following the syntax described in the the section `Text formatting` of the OCaml manual.

Additional documentation pages may by attached to a package can be attached using the *Documentation Stanza*.

9.3 Building Documentation

Building the documentation using the `@doc` alias. Hence, all that is required to generate documentation for your project is building 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:

```
$ dune build @doc-private
```

But this libraries will not be in the main html listing above, since they do not belong to any particular package. But the generated html will still be found in `_build/default/_doc/_html/<library>`.

9.4 Documentation Stanza

Documentation pages will be automatically generated for from .ml and .mli files that include ocamldoc fragments. Additional manual pages may be attached to packages using the `documentation` stanza. These .mld files must contain text in the same syntax as ocamldoc comments.

```
(documentation (<optional-fields>)))
```

Where `<optional-fields>` are:

- `(package <name>)` the package this documentation should be attached to. If this absent, dune will try to infer it based on the location of the stanza.
- `(mld_files <arg>)` where `<arg>` field follows the *Ordered set language*. This is a set of extension-less, mld file base names that are attached to the package. Where `:standard` refers to all the `.mld` files in the stanza's directory.

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. This is the page that will be linked from the main package listing. If you omit writing an `index.mld`, dune will generate one with the entry modules for your package. But this generated will not be installed.

All mld files attached to a package will be included in the generated `.install` file for that package, and hence will be installed by opam.

9.4.1 Examples

This stanza use 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 excluding `wip.mld` in the current directory to the inferred package:

```
(documentation
 ((mld_files (:standard \ wip))))
```

CHAPTER 10

Installing packages on the system

Installation is the process of copying freshly built libraries, binaries and other files from the build directory to the system. Dune offers two way 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 will describe how to define a package, how to "attach" various elements to it and how to proceed with installing it on the system.

10.1 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 to release two packages with the same name.

In older projects using Dune, packages were defined by the presence of a file called `<package-name>.opam` at the root of the project. However, it is 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.

10.2 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 sub-section 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 is opam who sets this directory. When calling `dune install`, the installation directory is either guessed or can be manually specified by the user. This is described more in detail in the last section of this page.

10.2.1 Libraries

In order to attach a library to a package all you need to do is 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 the public name of a library 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.

10.2.2 Executables

Similarly 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. The name that goes in there is the name under which the executables will be available through the `PATH` once installed, i.e. the name users will need to type in there shell to execute the program. Because Dune cannot guess which package an executable is part of from its public name, you also need 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.

10.2.3 Other files

For all other kinds of elements, you need to attach them manually via an `install` stanza. The install stanza takes three informations:

- the list of files the install
- the package to attach these files to. 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/mypacakge`.

The following sections are available:

- `lib` installs to `<prefix>/lib/<pkgname>/`
- `lib_root` installs to `<prefix>/lib/`
- **`libexec` installs to `<prefix>/lib/<pkgname>/` with the** executable bit set
- **`libexec_root` installs to `<prefix>/lib/` with the executable** bit set
- `bin` installs to `<prefix>/bin/` with the executable bit set
- `sbin` installs to `<prefix>/sbin/` with the executable bit set
- `toplevel` installs to `<prefix>/lib/toplevel/`
- `share` installs to `<prefix>/share/<pkgname>/`
- `share_root` installs to `<prefix>/share/`
- `etc` installs to `<prefix>/etc/<pkgname>/`
- `doc` installs to `<prefix>/doc/<pkgname>/`
- **`stublibs` installs to `<prefix>/lib/stublibs/` with the** executable bit set
- **`man` installs relative to `<prefix>/man` with the destination** directory extracted from the extension of the source file (so that installing `foo.1` is equivalent to a destination of `man1/foo.1`)
- **`misc` requires files to specify an absolute destination, and the** user will be prompted before the installation when it is done via opam. Only use this for advanced cases.

Normally, Dune uses the basename of the file to install to determine the name of the file once installed. However, you can change that fact 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)))
```

10.3 Installing a package

10.3.1 Via opam

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

10.3.2 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 the packages available in the workspace. When several build contexts are specified via a **'dune-workspace'_** file, the installation will be performed in all the build contexts.

10.3.3 Destination directory

The `<prefix>` directory is determined as follows for a given build context:

1. if an explicit `--prefix <path>` argument is passed, use this path
2. if `opam` is present in the `PATH` and is configured, use the output of `opam config var prefix`
3. otherwise, take the parent of the directory where `ocamlc` was found.

As an exception to this rule, library files might be copied to a different location. The reason for this is that they often need to be copied to a particular location for the various build system used in OCaml projects to find them and this location might be different from `<prefix>/lib` on some systems.

Historically, the location where to store OCaml library files was configured through findlib and the `ocamlfind` command line tool was used to both install these files and locate them. Many Linux distributions or other packaging systems are using this mechanism to setup where OCaml library files should be copied.

As a result, if none of `--libdir` and `--prefix` is passed to `dune install` and `ocamlfind` is present in the `PATH`, then library files will be copied to the directory reported by `ocamlfind printconf destdir`. This ensures that `dune install` can be used without opam. When using opam, `ocamlfind` is configured to point to the opam directory, so this rule makes no difference.

Note that `--prefix` and `--libdir` are only supported if a single build context is in use.

CHAPTER 11

Usage

This section describe usage of dune from the shell.

11.1 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. `dune init` can be used to quickly add new projects, libraries, tests, or executables without having to manually create dune files, or it can be composed to programmatically generate parts of a multi-component project.

11.1.1 Initializing a project

To initialize a new `dune` project that uses the `base` and `cmdliner`, libraries and supports inline tests, you can run

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

This will create a new directory called `myproj` including sub directories 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.

11.1.2 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)))
```

11.1.3 Initializing a library

To create a new directory `src`, initialized as a library, can run:

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

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

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

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

11.2 Finding the root

11.2.1 dune-workspace

The root of the current workspace is determined by looking up a `dune-workspace` or `dune-project` file in the current directory and parent directories.

`dune` prints out the root when starting if it is not the current directory:

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

More precisely, it will choose the outermost ancestor directory containing a `dune-workspace` file as root. For instance if you are in `/home/me/code/myproject/src`, then dune will look for all these files in order:

- `/dune-workspace`
- `/home/dune-workspace`
- `/home/me/dune-workspace`
- `/home/me/code/dune-workspace`
- `/home/me/code/myproject/dune-workspace`
- `/home/me/code/myproject/src/dune-workspace`

The first entry to match in this list will determine the root. In practice this means that if you nest your workspaces, dune will always use the outermost one.

In addition to determining the root, `dune` will read this file as to setup the configuration of the workspace unless the `--workspace` command line option is used. See the section *Workspace configuration* for the syntax of this file.

The `Entering directory` message can be suppressed with the `--no-print-directory` command line option (as in GNU make).

11.2.2 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.

11.2.3 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.

11.3 Interpretation of targets

This section describes how `dune` interprets the targets given on the command line. When no targets are specified, `dune` builds the `default` alias, see *Default alias* for more details.

11.3.1 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:

- `.merlin` files
- `<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` will expand it to the corresponding target in all the build contexts where it knows how to build it. When using `--verbose`, It prints out the actual set of targets when 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
```

11.3.2 Aliases

Targets starting with a `@` are interpreted as aliases. For instance `@src/runtest` means the alias `runtest` in all descendant of `src` in all build contexts where it is defined. If you want to refer to a target starting with a `@`, simply write: `./@foo`.

To build and run the tests for a particular build context, use `@_build/default/runtest` instead.

So for instance:

- `dune build @_build/foo/runtest` will run the tests only for the `foo` build context
- `dune build @runtest` will run the tests for all build contexts

You can also build an alias non-recursively by using `@@` instead of `@`. For instance to run tests only from the current directory:

```
dune build @@runtest
```

11.3.3 Default alias

When no targets are given to `dune build`, it builds the special `default` alias. Effectively `dune build` is equivalent to:

```
dune build @@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 install)))
```

Which means that by default `dune build` will build everything that is installable.

When using a directory as a target, it will be interpreted as building the default target in the directory. The directory must exist in the source tree.

```
dune build dir
```

Is equivalent to:

```
dune build @@dir/default
```

11.3.4 Built-in Aliases

There's a few aliases that dune automatically creates for the user

- `default` - this alias includes all the targets that dune will build if a target isn't specified, i.e. `$ dune build`. By default, this is set to the `install` alias.
- `runtest` - this is the alias to run all the tests, building them if necessary.
- `install` - build all public artifacts - those that will be installed.
- `doc` - build documentation for public libraries.
- `doc-private` - build documentation for all libraries - public & private.
- `lint` - run linting tools.
- `all` - build all available targets in a directory and installable artifacts defined in that directory.
- `check` - This alias will build the minimal set of targets required for tooling support. Essentially, this is `.cmi`, `.cmt`, `.cmti`, and .merlin files.

11.4 Finding external libraries

When a library is not available in the workspace, dune will look it up in the installed world, and expect it to be already compiled.

It looks up external libraries using a specific list of search paths. A list of search paths is specific to a given build context and is determined as follow:

1. if the `ocamlfind` is present in the `PATH` of the context, use each line in the output of `ocamlfind printconf path` as a search path
2. otherwise, if `opam` is present in the `PATH`, use the output of `opam config var lib`
3. otherwise, take the directory where `ocamlc` was found, and append `../lib` to it. For instance if `ocamlc` is found in `/usr/bin`, use `/usr/lib`

11.5 Running tests

There are two ways to run tests:

- `dune build @runtest`
- `dune runtest`

The two commands are equivalent. They will run all the tests defined in the current directory and its children recursively. You can also run the tests in a specific sub-directory and its children by using:

- `dune build @foo/bar/runtest`
- `dune runtest foo/bar`

11.6 Watch mode

The `dune build` and `dune runtest` commands support a `-w` (or `--watch`) flag. When it is 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.

11.7 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 will search (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.

11.7.1 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.

11.8 Restricting the set of packages

You can 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 is not in the list given to `dune`.

11.9 Invocation from opam

You should set the `build:` field of your `<package>.opam` file as follows:

```
build: [
  ["dune" "subst"] {pinned}
  ["dune" "build" "-p" name "-j" jobs]
]
```

`-p pkg` is a shorthand for `--root . --only-packages pkg --profile release --default-target @install`. `-p` is the short version of `--for-release-of-packages`.

This has the following effects:

- it tells dune to build everything that is installable and to ignore packages other than `name` defined in your project
- it sets the root to prevent dune from looking it up
- it silently ignores all rules with `(mode promote)`
- it sets the build profile to `release`
- it uses whatever concurrency option opam provides
- it sets the default target to `@install` rather than `@@default`

Note that `name` and `jobs` are variables expanded by opam. `name` expands to the package name and `jobs` to the number of jobs available to build the package.

11.10 Tests

To setup the building and running of tests in opam, add this line to your `<package>.opam` file:

```
build: [
  (* Previous lines here... *)
  ["dune" "runtest" "-p" name "-j" jobs] {with-test}
]
```

11.11 Workspace configuration

By default, a workspace has only one build context named `default` which correspond 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 is good practice to write a `dune-workspace.dev` in your project with all the version of OCaml your projects support. This way developers can tests that the code builds with all version of OCaml by simply running:

```
$ dune build --workspace dune-workspace.dev @all @runtest
```

11.11.1 dune-workspace

The `dune-workspace` file uses the S-expression syntax. This is what a typical `dune-workspace` file looks like:

```
(lang dune 1.0)
(context (opam (switch 4.02.3)))
(context (opam (switch 4.03.0)))
(context (opam (switch 4.04.0)))
```

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 1.0)
(context default)
```

This allows you to use an empty `dune-workspace` file to mark the root of your project.

profile

The build profile can be selected in the `dune-workspace` file by write a `(profile ...)` stanza. For instance:

```
(profile release)
```

Note that the command line option `--profile` has precedence over this stanza.

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 dune's *env* stanza.

context

The `(context ...)` stanza declares a build context. The argument can be either `default` or `(default)` for the default build context or can be the description of an opam switch, as follows:

```
(context (opam (switch <opam-switch-name>)
               <optional-fields>))
```

`<optional-fields>` are:

- `(name <name>)` is the name of the subdirectory of `_build` where the artifacts for this build context will be stored
- `(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)` instructs dune to use this build context for merlin
- `(profile <profile>)` to set a different profile for a build context. This has precedence over the command line option `--profile`
- `(env <env>)` to set the environment for a particular context. This is of higher precedence than the toplevel `env` stanza in the workspace file. This field the same options as the *env* stanza.
- `(toolchain <findlib_coolchain>)` set findlib toolchain for the context.
- `(host <host_context>)` choose a different context to build binaries that are meant to be executed on the host machine, such as preprocessors.

Both `(default ...)` and `(opam ...)` accept a `targets` field in order to setup cross compilation. See *Cross Compilation* for more information.

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 contexts for merlin by adding the field `(merlin)` to this context.

11.12 Distributing Projects

Dune provides support for building and installing your project. However it doesn't provide helpers for distributing it. It is 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`

And that if your project contains several packages, then all the package names must be prefixed by the shortest one.

11.13 Watermarking

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

This is especially interesting for the `VERSION` watermark, which gets replaced by the version obtained from the vcs. For instance if you are using git, dune-release invokes this command to find out the version:

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

Projects using dune usually only need dune-release for creating and publishing releases. However they might still want to substitute the watermarks when the package is pinned by the user. To help with this, dune provides the `subst` sub-command.

11.14 dune subst

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

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

- `NAME`, the name of the project
- `VERSION`, output of `git describe --always --dirty`
- `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 name of the project 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 in particular 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.

11.15 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
```

CHAPTER 12

Advanced topics

This section describes some details of dune for advanced users.

12.1 META file generation

Dune uses `META` files from the findlib library manager in order to interoperate with the rest of the world when installing libraries. It is able to generate them automatically. However, for the rare cases where you would need a specific `META` file, or to ease the transition of a project to dune, it is allowed to write/generate a specific one.

In order to do that, write or setup a rule to generate a `META.<package>.template` file in the same directory as the `<package>.opam` file. Dune will generate a `META.<package>` file from the `META.<package>.template` file by replacing lines of the form `# DUNE_GEN` by the contents of the `META` it would normally generate.

For instance if you want to extend the `META` file generated by dune you can write the following `META.foo.template` file:

```
# DUNE_GEN
blah = "..."
```

12.2 Findlib integration and limitations

Dune uses `META` files to support external libraries. However, it doesn't export the full power of findlib to the user, and especially it doesn't let the user specify *predicates*.

The reason for this limitation is that so far they haven't been needed, and adding full support for them would complicate things quite a lot. In particular, complex `META` files are often hand-written and the various features they offer are only available once the package is installed, which goes against the root ideas dune is built on.

In practice, dune interprets `META` files assuming the following set of predicates:

- `mt`: what this means is that using a library that can be used with or without threads with dune will force the threaded version

- `mt_posix`: forces the use of posix threads rather than VM threads. VM threads are deprecated and are likely to go away soon
- `ppx_driver`: when a library acts differently depending on whether it is linked as part of a driver or meant to add a `-ppx` argument to the compiler, choose the former behavior

12.3 Dynamic loading of packages

Dune supports the `findlib.dynload` package from findlib that allows to dynamically load packages and their dependencies (using OCaml Dynlink module). So adding the ability for an application to have plugins just requires to add `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 ...)
)
```

Then you could use in your application `Fl_dynload.load_packages l` that will 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 need to link to your library:

```
(library
  (name mytool_plugin_a)
  (public_name mytool-plugin-a)
  (libraries mytool)
)
```

By choosing some 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 = Fl_package_base.list_packages () in
  let pkgs =
    List.filter
      (fun pkg -> 14 <= String.length pkg && String.sub pkg 0 14 = "mytool-plugin-")
      pkgs
  in
  Fl_dynload.load_packages pkgs
```

12.4 Cross Compilation

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

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

- `native` which means using the native tools that can build binaries that 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, when implicitly added `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.

12.4.1 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 will 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})))
```

12.5 Classical ppx

classical ppx refers to running ppx using the -ppx compiler option, which is composed using Findlib. Even though this is useful to run some (usually old) ppx's which don't support drivers, dune does not support preprocessing with ppx this way. but a workaround exists using the ppxfind tool.

12.6 Profiling dune

If `--trace-file FILE` is passed, dune will write detailed data about internal operations, such as the timing of commands that are run by dune.

The format is compatible with Catapult trace-viewer. In particular, these files can be loaded into Chromium's `chrome://tracing`. Note that the exact format is subject to change between versions.

12.7 Implicit Transitive Deps

By default, dune allows transitive dependencies of dependencies to be used directly when compiling OCaml. However, this setting can be controlled per project. It can be disabled by adding the `(implicit_transitive_deps false)` to the `dune-project` file.

Once this setting is added, all dependencies that are directly used by a library or an executable must be directly added in the `libraries` field. We recommend users to experiment with this mode and report any problems. The goal is to make this the default mode eventually.

Note that you must use `threads.posix` instead of `threads` when using this mode. This is not an important limitation as `threads.vm` are deprecated anyways.

12.8 Name Mangling of Executables

Executables are made of compilation units whose names may collide with the compilation units of libraries. To avoid this possibility, dune prefixes these compilation unit names with `Dune__exe__`. This is entirely transparent to users except for when such executables are debugged. In which case the mangled names will be visible in the debugger.

Starting from dune 1.11, the `(wrapped_executables <bool>)` option is available to turn on/off name mangling for executables on a per project basis.

Starting from dune 2.0, dune mangles compilation units of executables by default. However, this can still be turned off using `(wrapped_executables false)`

12.9 Explicit JS mode

By default, Javascript targets are defined for every bytecode executable that dune knows about. This is not very precise and does not interact well with the `@all` alias (eg, the `@all` alias will try to build JS targets corresponding to every `test` stanza). In order to better control the compilation of JS targets, this behaviour can be turned off by using `(explicit_js_mode)` in the `dune-project` file.

When explicit JS mode is enabled, an explicit *js* mode needs to be added to the `(modes ...)` field of executables in order to trigger JS compilation. Explicit JS targets declared like this will be attached to the `@all` alias.

Starting from dune 2.0 this new behaviour will be the default and JS compilation of binaries will need to be explicitly declared.

12.10 Dialects

A dialect is 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.

The extensions are unique among all dialects of a given project, so that a given extension can be mapped back to the corresponding dialect.

A dialect 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.

Similarly, a dialect can specify a custom formatter to implement the `@fmt` alias, see *Automatic formatting*.

When not using a custom syntax or formatting action, a dialect is nothing but a way to specify custom file extensions for OCaml code.

12.10.1 Defining a dialect

A dialect can be defined by adding the following to the `dune-project` file:

```
(dialect
 (name <name>)
 (implementation
  (extension <string>)
  <optional fields>)
 (interface
  (extension <string>)
  <optional fields>))
```

`<name>` is the name of the dialect being defined. It must be unique in a given project.

`(extension <string>)` specifies the file extension used for this dialect, for interfaces and implementations. The extension string must not contain any dots, and be unique in a given project.

`<optional fields>` are:

- `(preprocess <action>)` is the action to run to produce a valid OCaml abstract syntax tree. It is expected to read the file given in the variable named `input-file` and output a *binary* abstract syntax tree on its standard output. See *Preprocessing with actions* for more information.

  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.

- `(format <action>)` is the action to run 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. For more information. See *Automatic formatting* for more information.

  If the field is not present, then if `(preprocess <action>)` is not present (so that the dialect consists of valid OCaml code), then by default the dialect will be formatted as any other OCaml code. Otherwise no special formatting will be done.

CHAPTER 13

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, 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. You do not need to install anything to use configurator.

13.1 Usage

We'll describe configurator with a simple example. Everything else can be easily learned by studying configurator's API.

To use configurator, we 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()
```

(continues on next page)

(continued from previous page)

```
{
  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)
 (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`.

13.2 Upgrading from the old Configurator

The old configurator is the independent configurator opam package. It is 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 removed.
- The library name `configurator` should be changed `dune.configurator`.
- The `-ocamlc` flag in rules that run configurator scripts should be removed. This information is now passed automatically by dune.
- The new configurator API is versioned explicitly. The version that is 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
```

CHAPTER 14

Menhir

To use menhir in a dune project, the language version should be selected in the `dune-project` file. For example:

```
(using menhir 2.0)
```

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 to the project file once a menhir stanza is used anywhere.

14.1 Basic Usage

The basic form for defining menhir parsers (analogous to ocamlyacc) is:

```
(menhir
 (modules <parser1> <parser2> ...))
```

14.2 Modular Menhir

Modular parsers can be defined by adding a `merge_into` field. This correspond to the `--base` command line option of `menhir`. With this option, a single parser named `base_name` is generated.

```
(menhir
 (merge_into <base_name>)
 (modules <parser1> <parser2> ...))
```

14.3 Flags

Extra flags can be passed to menhir using the `flags` flag:

```
(menhir
 (flags <option1> <option2> ...)
 (modules <parser1> <parser2> ...))
```

14.4 `--infer` mode

Menhir language 2.0 automatically enables using menhir with type inference. This ability can also be manually controlled with the `infer` field manually.

```
(menhir
  (infer false)
  (modules <parser1> <parser2> ...))
```

14.5 cmly targets

Menhir supports writing the grammar and automaton to `.cmly` file. Therefore, if this is flag is passed to menhir, dune will know to introduce a `.cmly` target for the module.

CHAPTER 15

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
```

15.1 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))
```

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.

15.2 `js_of_ocaml` field

In `library` and `executables` 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`. This field supports `(:include ...)` forms
- `(javascript_files (<files-list>))` to specify `js_of_ocaml` JavaScript runtime files.

`<flags>` is specified in the *Ordered set language*.

The default value for `(flags ...)` depends on the selected build profile. The build profile `dev` (the default) will enable sourcemap and the pretty JavaScript output.

15.3 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. There is currently no other way to control this behaviour.

CHAPTER 16

opam

opam is the official package manager for OCaml, and dune offers some integration with it.

16.1 Generating opam files

Dune is able to 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 true)
```

Dune uses the following global fields to set the metadata for all packages defined in the project:

- `(license <name>)` - Specifies the license of the project, ideally as an identifier from the SPDX License List
- `(authors <authors>)` - A list of authors
- `(maintainers <maintainers>)` - A list of maintainers
- `(source <source>)` - where the source is specified two ways: `(github <user/repo>)` or `(uri <uri>)`
- `(bug_reports <url>)` - Where to report bugs. This defaults to the GitHub issue tracker if the source is specified as a GitHub repository
- `(homepage <url>)` - The homepage of the project
- `(documentation <url>)` - Where the documentation is hosted

Package specific information is specified in the `(package <package>)` stanza. It contains the following fields:

- `(name <string>)` is the name of the package. This must be specified.
- `(synopsis <string>)` is a short package description
- `(description <string>)` is a longer package description
- `(depends <dep-specification>)` are package dependencies

- `(conflicts <dep-specification)` are package conflicts
- `(depopts <dep-specification)` are optional package dependencies
- `(tags <tags>)` are the list of tags for the package

The list of dependencies `<dep-specification>` is modeled after opam's own language: The syntax is as a list of the following elements:

```
op := '=' | '<' | '>' | '<>' | '>=' | '<='

stage := :with_test | :build | :dev

constr := (<op> <version>)

logop := or | and

dep := (name <stage>)
     | (name <constr>)
     | (name (<logop> (<stage> | <constr>)*))

dep-specification = dep+
```

Here's a complete example of a dune file with opam metadata specification:

```
(lang dune 1.10)
(name cohttp)
(source (github mirage/ocaml-cohttp))
(license ISC)
(authors "Anil Madhavapeddy" "Rudi Grinberg")
(maintainers "team@mirage.org")

(package
 (name cohttp)
 (synopsis "An OCaml library for HTTP clients and servers")
 (description "A longer description")
 (depends
  (alcotest :with-test)
  (dune (and :build (> 1.5)))
  (foo (and :dev (> 1.5) (< 2.0)))
  (uri (>= 1.9.0))
  (uri (< 2.0.0))
  (fieldslib (> v0.12))
  (fieldslib (< v0.13))))

(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))
  (async (< v0.12))))
```

16.1.1 Opam Template

A user may want to manually fill in some field in the opam file. In these situations, dune provides an escape hatch in the form of a user written opam template. An opam template must be named `<package>.opam.template` and should be a syntactically valid opam file. Any field defined in this template file will be taken as is by dune and never overwritten.

Note the template file cannot be generated by a rule and must be available in the source tree.

CHAPTER 17

Virtual Libraries & Variants

Virtual libraries correspond to dune's ability to compile parameterized 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 and compile against the virtual library and 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.

17.1 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.

17.2 Implementation

An implementation for a library is defined as:

```
(library
 (name clock_unix)
 ;; 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 does not 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))
```

17.3 Variants

This feature is still under development and may change with new dune releases. You need to write `(using library_variants 0.2)` in your `dune-project` file to unlock it.

When building a binary, implementations can be selected using a set of variants rather than individually specifying implementations.

An example where this is useful is providing JavaScript implementation. It would be tedious to select the JS implementation for every single virtual library. Instead, such implementations could select a `js` variant. Here's the syntax:

```
(executable
 (name foo)
 (libraries time filesystem)
 (variants js))
```

An implementation can specify which variant it corresponds to using the `variant` option. Say for example that `time` is a virtual library. Its JS implementation would have the following configuration:

```
(library
 (name time-js)
 (implements time)
 (variant js))
```

The list of available variants is computed while building the virtual library. This means only variant implementations that are part of the same project are implicitely taken into account. It's possible to declare an external implementation by using the *external_variant* stanza in the virtual library scope.

```
(external_variant
 (variant foo)
```

(continues on next page)

(continued from previous page)

```
 (implementation lib-foo)
 (virtual_library vlib))
```

This will add *lib-foo* to the list of known implementations of *vlib*.

17.4 Default implementation

This feature is also guarded by `(using library_variants ...)`.

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))
```

17.5 Limitations

The current implementation of virtual libraries suffers from a few limitations. Some of these are temporary.

- It is not possible to link more than one implementation for the same virtual library in one executable.
- It is 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's not possible to load virtual 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 is not possible for a library to be both virtual and implement another library. This isn't very useful, but technically, it could be used to create partial implementations. It is possible to lift this restriction if there's enough demand for this.

CHAPTER 18

Automatic formatting

Dune can be set up to run automatic formatters for source code.

It can use ocamlformat to format OCaml source code (`*.ml` and `*.mli` files) and refmt to format Reason source code (`*.re` and `*.rei` files).

Furthermore it can be used to format code of any defined dialect (see *Dialects*).

18.1 Enabling automatic formatting

This feature is enabled by adding the following to the `dune-project` file:

```
(using fmt 1.2)
```

18.2 Formatting a project

When this feature is active, an alias named `fmt` is defined. When built, 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"
```

It is then possible to accept the correction by calling `dune promote` to replace the source files by the corrected versions.

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

As usual with promotion, it is possible to combine these two steps by running `dune build @fmt --auto-promote`.

18.3 Only enabling it for certain languages

By default, formatting will be enabled for all languages and dialects present in the project that dune knows about. This is not always desirable, for example if in a mixed Reason/OCaml project, one only wants to format the Reason files to avoid pulling `ocamlformat` as a dependency.

In these cases, it is possible to use the `enabled_for` argument to restrict the languages that are considered for formatting.

```
(using fmt 1.2 (enabled_for reason))
```

18.4 Version history

18.4.1 1.2

- Format *Dialects*.

18.4.2 1.1

- Format Dune files.

18.4.3 1.0

- Format OCaml (using ocamlformat) and Reason (using refmt) source code.

CHAPTER 19

Coq

Dune is also able to build Coq developments. A Coq project is a mix of Coq `.v` files and (optionally) OCaml libraries linking to the Coq API (in which case we say the project is a *Coq plugin*). To enable Coq support in a dune project, the language version should be selected in the `dune-project` file. For example:

```
(using coq 0.1)
```

This will enable support for the `coq.theory` stanza in the current project. If the language version is absent, dune will automatically add this line with the latest Coq version to the project file once a `(coq.theory ...)` stanza is used anywhere.

19.1 Basic Usage

The basic form for defining Coq libraries is very similar to the OCaml form:

```
(coq.theory
 (name <module_prefix>)
 (public_name <package.lib_name>)
 (synopsis <text>)
 (modules <ordered_set_lang>)
 (libraries <ocaml_libraries>)
 (flags <coq_flags>))
```

The stanza will build all *.v* files on the given directory. The semantics of fields is:

- `<module_prefix>>` will be used as the default Coq library prefix `-R`,
- the `modules` field does allow to constraint the set of modules included in the library, similarly to its OCaml counterpart,
- `public_name` will make Dune generate install rules for the *.vo* files; files will be installed in `lib/coq/user-contrib/<module_prefix>`, as customary in the make-based Coq package eco-system. For compatibility, we also installs the *.cmxs* files appearing in <*ocaml-librarie*> under the *user-contrib* prefix.
- `<coq_flags>` will be passed to `coqc`,

- the path to installed locations of `<ocaml_libraries>` will be passed to `coqdep` and `coqc` using Coq's `-I` flag; this allows for a Coq library to depend on a ML plugin.

19.2 Preprocessing with coqpp

Coq plugin writers usually need to write `.mlg` files to extend Coq grammar. Such files are pre-processed with *coqpp*; to help plugin writers avoid boilerplate we provide a *(coqpp ...)* stanza:

```
(coq.pp (modules <mlg_list>))
```

which for each `g_mod` in ‘‘<mlg_list>‘‘‘is equivalent to:

```
(rule
 (targets g_mod.ml)
 (deps (:mlg-file g_mod.mlg))
 (action (run coqpp %{mlg-file})))
```

19.3 Recursive Qualification of Modules

If you add:

```
(include_subdirs qualified)
```

to a `dune` file, Dune will to consider that all the modules in their directory and sub-directories, adding a prefix to the module name in the usual Coq style for sub-directories. For example, file `A/b/C.v` will be module `A.b.C`.

19.4 Limitations

- composition and scoping of Coq libraries is still not possible. For now, libraries are located using Coq's built-in library management,
- .v always depend on the native version of a plugin,
- a `foo.mlpack` file must the present for locally defined plugins to work, this is a limitation of coqdep,

CHAPTER 20

FAQ

20.1 Why do many dune projects contain a Makefile?

Many dune projects contain a toplevel *Makefile*. It is 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 are not part of the build and *make <blah>* is a good way to provide them
3. *make* is shorter to type than *dune build @install*

20.2 How to add a configure step to a dune project?

The with-configure-step example shows one way to do it which preserves composability; i.e. it doesn't require manually running *./configure* script when working on multiple projects at the same time.

20.3 Can I use topkg with dune?

It's possible using the topkg-jbuilder but it's not recommended. dune-release subsumes topkg-jbuilder and is specifically tailored to dune projects.

20.4 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 tool for publishing dune packages.

20.5 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 depending on dune. It is therefore a useful reference to search through to find different approaches to constructing build rules.

20.6 What is Jenga?

jenga is a build system developed by Jane Street mainly for internal use. It was never usable outside of Jane Street, and hence 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.

20.7 How to make warnings non-fatal?

`jbuilder` used to display warnings, but most of them would not stop the build. But `dune` makes all warnings fatal by default. This can be a challenge when porting a codebase to `dune`. There are two ways to 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.
- you can change the flags that are used by the `dev` profile by adding the following stanza to a `dune` file:

```
(env
  (dev
    (flags (:standard -warn-error -A))))
```

CHAPTER 21

Known Issues

21.1 mli only modules

These are supported, however using them might cause make it impossible for non-dune users to use your library. We tried to use them for some internal module generated by dune and it broke the build of projects not using dune:

https://github.com/ocaml/dune/issues/567

So, while they are supported, you should be careful where you use them. Using a *.ml* only module is still preferable.

CHAPTER 22

Migration

Dune was initially called Jbuilder. Up to mid-2018, the package was still called *jbuilder* which only installed a *jbuilder* binary. This document explain how the migration to Dune will happen.

22.1 Timeline

The general idea is that the migration is gradual and existing Jbuilder projects don't need to be updated all at once. We encourage users to switch their development repositories and continue their usual release cycle. There is no need to re-release existing packages just to switch to Dune immediately.

The plan is as follows:

22.1.1 July 2018: release of Dune 1.0.0

First release of the opam package *dune*. The *jbuilder* package becomes a transitional package that depends on *dune*.

The *dune* package installs two binaries: *dune* and *jbuilder*. These two binaries are exactly the same and they work on both Jbuilder and Dune projects. Additionally they recognize both Jbuilder and Dune configuration files. The new Dune configuration files are described later in this document.

22.1.2 January 2019: deprecation of Jbuilder

At this point, the *jbuilder* binary emits a warning on every startup inviting users to switch to *dune*. When encountering *jbuild* or other Jbuilder configuration files, both binaries emit a warning. The rest is unchanged.

During this period, it makes sense for projects to do new releases just to switch to Dune if none of their existing releases is using Dune.

22.1.3 July 2019: support for Jbuilder is dropped

jbuilder is now a dummy executable that always exit with an error message on startup. *dune* no longer reads *jbuild* or other Jbuidler configuration files but still prints a warning when encountering them.

At this point, a conflict with newer versions of *dune* will be added to all opam packages that rely on the *jbuilder* binary or Jbuilder configuration files.

22.1.4 January 2020: the jbuilder binary goes away

The *dune* package no longer installs a *jbuilder* binary. The rest is unchanged.

22.1.5 Distant future

Once we are sure there are no more *jbuild* files out there, Dune will completely ignore *jbuild* and other Jbuilder configuration files.

22.2 Check List

This section is a concise list of migration tasks that will be required to transition from jbuilder to dune.

22.2.1 New configuration files

Until July 2019, *dune* will still read *jbuild* and other Jbuilder configuration files. There is no change in these files.

However, based on the experience acquired since the first release of Jbuilder, we made a few changes in the configuration files read by Dune. The most notable ones are the following:

- *jbuild* files are renamed simply *dune*
- projects now have a *dune-project* file at their root
- *jbuild-ignore* files are replaced by *ignored_subdirs* stanzas in *dune* files
- *jbuild-workspace* are replaced by *dune-workspace* files
- *jbuild-workspace<suffix>* files no longer mean anything

Following are detailed explanation of the differences between the Jbuilder configuration files and the Dune ones.

22.2.2 dune-project files

These are a new kind of file. With Jbuilder, projects used to be identified by the presence of at least one *<package>.opam* file in a directory. This will still be supported until July 2019, however as Jbuilder evolved it became clear that we needed project files, so Dune introduces *dune-project* files to mark the root of projects.

Eventually, we are hoping that Dune will generate opam files. So users will only have to write a *dune-project* file.

The purpose of this file is to:

- delimit projects in larger workspaces
- set a few project-wide parameters, such as the name, the version of the Dune language in use or specification of extra features (plugins) used in the project

Eventually, for users who wish to do so it should be possible to centralize all the configuration of a project in this file.

22.2.3 dune files

These are the same as *jbuild* files.

22.2.4 dune-workspace

These are the same as *jbuild-workspace* files.

When looking for the root of the workspace, Jbuilder also looks for files whose name start with *jbuild-workspace*, such as *jbuild-workspace.in*. This rule will be kept until July 2019, however it is not preserved for *dune-workspace* files. I.e. a *dune-workspace.in* file means nothing.

This rule was only useful when we didn't have project files.

22.2.5 Variable Syntax

`${foo} and $(foo)` are no longer valid variable syntax in dune files. Variables are defined as `%{foo}`. This change is done to simplify interoperability with bash commands which also use the `${foo}` syntax.

22.2.6 `(files_recursively_in ..)` is removed

The `files_recursively_in` dependency specification is invalid in dune files. A *source_tree* stanza has been introduced to reflect the actual function of this stanza.

22.2.7 Escape Sequences

Invalid escape sequences of the form `\x` where `x` is a character other than `[0-9]`, `x`, `n`, `r`, `t`, `b` are not allowed in dune files.

22.2.8 Comments Syntax

Block comments of the form `#| ... |#` and comments of the form `#;` are not supported in dune files.

22.2.9 Renamed Variables

All existing variables have been lowercased for consistency. Other variables have always been renamed. Refer to this table for details:

Jbuild	Dune
${@}	%{targets}
${^}	%{deps}
${path:file}	%{dep:file}
${SCOPE_ROOT}	%{project_root}
${ROOT}	%{workspace_root}
${findlib:..}	%{lib:..}
${CPP}	%{cpp}
${CC}	%{cc}
${CXX}	%{cxx}
${OCAML}	%{ocaml}
${OCAMLC}	%{ocamlc}
${OCAMLOPT}	%{ocamlopt}
${ARCH_SIXTYFOUR}	%{arch_sixtyfour}
${MAKE}	%{make}

22.2.10 Removed Variables

`${path-no-dep:file}` and `${<}` have been removed.

A named dependency should be used instead of `${<}`. For instance the following jbuild file:

```
(alias
 ((name    runtest)
  (deps    (input))
  (action (run ./test.exe %{<}))))
```

should be rewritten to the following dune file:

```
(alias
 (name    runtest)
 (deps    (:x input))
 (action (run ./test.exe %{x})))
```

22.2.11 `# JBUILDER_GEN` renamed

`# DUNE_GEN` should be used instead of `# JBUILDER_GEN` in META templates.