The OCamlbuild manual

Table of Contents

OCamlbuild's job is to determine the sequence of calls to the compiler — with the right set of command-line flags — needed to build your OCaml-centric software project.

1. Overview

1.1. Is OCamlbuild well-suited for your project?

OCamlbuild is extremely convenient to use for simple projects. If you have a small OCaml project (program or library), you could likely directly invoke `ocamlbuild` to automatically discover various source files and dependencies, and build executables, library archives or documentation with one-line commands. In simple cases you don't need to write a configuration file at all.

A few examples of quick `ocamlbuild` commands:

```
# builds a bytecode executable out of foo.ml and its local dependencies
ocamlbuild 'foo.byte'

# builds a native executable
ocamlbuild 'foo.native'

# builds a library archive from the modules listed (capitalized) in lib.mllib
ocamlbuild 'lib.cma'

# builds OCamldoc documentation from the modules listed in lib.odocl
ocamlbuild 'lib.docdir/index.html'

# enable a few ocamlfind packages and compile in debug mode
ocamlbuild -use-ocamlfind -pkgs 'lwt,react,yojson,sedlex.ppx' -tag 'debug'
'foo.native'
```

If repeating `-pkgs 'lwt,react,yojson,sedlex.ppx' -tag 'debug'` becomes bothersome, you can create a `_tags` file in the project directory with the content:

```
true: package(lwt), package(react), package(sedlex), package(yojson), debug
```

, and then just use:

```
ocamlbuild -use-ocamlfind 'foo.native'
```

OCamlbuild was designed as a generic build system (it is in fact not OCaml-specific), but also to be expressive enough to cover the specifics of the OCaml language that make writing good `Makefile`s difficult, such as the dreaded `units Foo and Bar make inconsistent assumptions about Baz` error.

1.2. Examples

We maintain a few self-contained examples of projects using various OCamlbuild features under `examples/`:

Table 1. Examples

Directory	Description
`01-simple`	A minimal example with self-contained OCaml code in the project directory.
`02-subdirs`	OCaml code in `lib/` and `src/` subdirectories. See <<Sec_Directories> for the details.
`03-packages`	A simple example of use of an Findlib package (`yojson`) in an OCamlbuild project. See Findlib-based packages introduction section for more details.
`04-library`	A simple example of using a `.mllib` file to easily build library archives (`.cma` in bytecode, `.cmxa` with native compilation). See Archives, documentation for more details.
`05-lex-yacc`	A simple program using `ocamllex` to generate a lexer, and Menhir to generate a parser (ocamlyacc would work as easily, but we recommend using Menhir instead which is just a better parser generator). See `ocamlyacc` and Menhir targets in Reference documentation for all parser-relevant options.

> There are many ways to integrate OCamlbuild in your project. The examples provided so far use a `Makefile` on top of OCamlbuild to provide the familiar `make; make install; make clean` interface to users, but you are free to do otherwise.

1.3. Pros, cons, and alternatives

1.3.1. Pros

- It 'Just Works' for most OCaml projects, with **minimal configuration** work on your part.
- It is designed from the scratch with **dynamic dependencies** in mind. Dynamic dependencies are the dependencies that are only determined during the build of the target, e.g., the local modules on which a source file depends, and are not explicitly listed in a configuration file. This avoids, for example, the dance of pre-generating or post-generating `.depends` files that is occasionally bothersome with `Makefile` projects, without requiring you to describe all local dependency relations manually either.

1.3.2. Cons

- Instead of a homegrown soon-to-become-Turing-complete configuration language, OCamlbuild made the choice of using **OCaml as its configuration language**, and many users dislike this choice. Most features of OCamlbuild can be controlled through the purely declarative `_tags` file whose structure is explained in detail in this documentation, but for more complex projects you will eventually need to create a `myocamlbuild.ml` to configure the tool through its OCaml library interface. We strive for a simple API and we document it, so that should be more pleasant than you expect.

- OCamlbuild is not the most efficient build system out there, and in particular its support for **build parallelization** is currently disappointing.

  > This could be solved with more engineering effort. OCamlbuild is maintained by volunteers, and your contributions are warmly welcome, see `CONTRIBUTING.adoc`.

  For now, OCamlbuild's default build rules will not scale to millions-of-lines codebases. It is however used in countless useful libraries and projects where this has not been a limitation.

1.3.3. Alternatives

- OCaml-Makefile, a generic set of Makefile rules for OCaml projects. Valuable if you want to build OCaml projects with `make`, without rewriting your own boilerplate from scratch.
- OMake, a generic build system that has been succesfully used by several relatively large OCaml projects.
- jenga, a build tool developed internally at Jane Street. The design is interestingly close to OCamlbuild's, but with large engineering efforts oriented towards building their huge internal codebase. There is no easy-to-use frontend layer provided to build OCaml projects (and little documentation), it's more of a build-your-own-build-system toolkit for now.
- obuild wants to be a really-simple build system with a declarative configuration language that has 80% the features, to cover most simple projects.

> The *Real World OCaml* textbook uses a tool named `corebuild`. This is in fact just a simple wrapper on top of `ocamlbuild` provided by the OCaml library named `core`, with some common options for `core` projects baked in.

1.4. Core concepts

1.4.1. Rules and targets

OCamlbuild knows about a set of *rules* to build programs. Rules are OCaml code that specifies on a high level how to build certain kinds of files, named *targets*, from some dependencies (statically known or dynamically discovered). For example, a built-in "%.ml → %.cmo" rule describes how to build any `.cmo` compilation unit file from the `.ml` of the same name; if you call `ocamlbuild foo.cmo`, it will either use `foo.ml` in your source directory or, if it doesn't exist, try to build it, for example from `foo.mll` or `foo.mly`.

OCamlbuild knows various targets to build all sorts of useful things: byte or native programs (`.byte`, `.native`), library archives (`.cma`, `.cmxa`, `.cmxs`), documentation (`.docdir/index.html`, `.docdir/man`), etc. We will detail these in the Reference section.

1.4.2. Tags and the `_tags` file

Tags are an abstraction layer designed to specify command-line flags in a declarative style. If you're invoking the compiler directly and wish to build a program with debug information enabled, you need to pass the `-g` flag to the compilation and linking step of the build process, but not during an initial syntactic preprocessing step (if any), when building `.cma` library archives, or when calling `ocamldoc`. With OCamlbuild, you can simply add the `debug` tag to your program's targets, and it will sort out when to insert the `-g` flag or not.

To attach tags to your OCamlbuild targets, you write them in a `_tags` file. Each line is of the form `foo: bar`, where `bar` is a list of tags, and `foo` is a filter that determines which targets `bar` applies to.

For example, the `_tags` file:

```
true: package(toto), package(tata)
<foo.*> or <bar.*>: debug
"strange.ml": rectypes
```

will make your whole project (`true` matches anything) depend on the Findlib packages `toto` and `tata`, compile modules `foo` and `bar` with debug information, and pass `-rectypes` when compiling `strange.ml` — but not `strange.mli`. For more detail, see the syntax of predicates in tags, and the set of built-in tags.

1.4.3. `myocamlbuild.ml`

The `_tags` file provides a convenient but limited interface to tune your project. For any more general purpose, we chose to use a configuration file directly written in OCaml, instead of reinventing a home-made configuration language — or using your shell as Make does. Code put in the `myocamlbuild.ml` file at the root of your project will be compiled and executed by `ocamlbuild` upon invocation.

For simple use cases, you should not have to write a `myocamlbuild.ml` file, except maybe to specify project-wide configuration options — similar to command-line options you would pass to OCamlbuild. But it also allows to define new rules and targets (for example to support a shiny new preprocessing program), to define new tags or refine the meaning of existing tags. We will cover these use-cases in the more advanced section Enriching OCamlbuild through plugins.

1.5. Working with a simple program

Simple OCaml projects often have a set of `.ml` and `.mli` files that provide useful modules depending on each other, and possibly a main file `myprog.ml` that contains the main program code.

```
mod1.ml
mod1.mli
mod2.ml
myprog.ml
```

You can build your program using either the bytecode compiler, with

```
ocamlbuild 'myprog.byte'
```

or the native compiler, with

```
ocamlbuild 'myprog.native'
```

Let's look at the organization of your source directory after this compilation command:

```
_build/
mod1.ml
mod1.mli
mod2.ml
myprog.byte -> _build/myprog.byte*
myprog.ml
```

OCamlbuild does all its work in a single `_build` directory, to help keep your source code directory tree clean. Targets are therefore built inside `_build`. It will generally add a symbolic link for the requested target in the user directory, but if a target does not appear after being built, chances are it is in `_build`.

1.6. Hygiene

OCamlbuild will proactively complain if some compiled files/build artifacts were left in the source directory:

```
# In the source directory of the previous example.
ocamlc -c 'mod2.ml'
ocamlbuild 'myprog.byte'
```

```
  SANITIZE: a total of 2 files that should probably not be in your
    source tree has been found. A script shell file
    "_build/sanitize.sh" is being created.  Check this script and
    run it to remove unwanted files or use other options (such as
    defining hygiene exceptions or using the -no-hygiene option).
  IMPORTANT: I cannot work with leftover compiled files.
  ERROR: Leftover OCaml compilation files:
    File mod2.cmo in . has suffix .cmo
    File mod2.cmi in . has suffix .cmi
  Exiting due to hygiene violations.
```

```
rm mod2.cm*
```

If this annoys you, it is possible to exclude some files from this hygiene checking by tagging them with the `precious` or `not_hygienic` tags, or to disable the check globally using the `-no-hygiene` command-line option.

The reason for this check is that leftover intermediate files can disrupt the way your build system works. OCamlbuild knows which target you need (library archives or program executables), and tries to build their dependencies, which first builds the dependencies of those dependencies, etc., until it eventually reaches your source files (the *inputs* of the build process). Everything present in the source directory is considered to be an input; if you keep old `.cmo` files in your source repository, OCamlbuild will not try to rebuild them from source files, but take them as references to produce the final targets, which is not what you want if they are stale.

1.7. Findlib-based packages

Your project will probably depend on external libraries as well. Let's assume they are provided via the Findlib tool `ocamlfind`. Suppose we depend on packages `tata` and `toto`. To tell OCamlbuild about them, use the tags `package(tata)` and `package(toto)`. You also need to tell OCamlbuild to enable support for `ocamlfind` by passing the `-use-ocamlfind` command-line option.

So you will have the following `_tags` file:

```
true: package(tata), package(toto)
```

and invoke compilation with

```
ocamlbuild -use-ocamlfind 'myprog.byte'
```

> Because of the popularity of Findlib you can expect to always invoke `ocamlbuild` with the `-use-ocamlfind` option. We will probably enable `-use-ocamlfind` by default in future versions of OCamlbuild, but in the meantime feel free to define a shell alias for convenience.

If you have a `myocamlbuild.ml` file (see Enriching OCamlbuild through plugins) at the root of your OCamlbuild project, you can use it to set this option, instead of using one command line parameter. Something like this:

```
open Ocamlbuild_plugin
let () =
  dispatch (function
    | Before_options ->
      Options.use_ocamlfind := true
    | _ -> ())
```

1.8. Syntax extensions

If you use syntax extensions distributed through `ocamlfind`, you can use them as any Findlib package, but you must also use the `syntax(…)` tag to indicate which preprocessor you use: `camlp4o`, `camlp4r`, `camlp5o`, etc.

```
true: syntax(camlp4o)
true: package(toto), package(blah.syntax)
```

In recent versions of OCamlbuild (since OCaml 4.01), you can also specify this using the `-syntax` command-line option:

```
ocamlbuild -use-ocamlfind -syntax 'camlp4o' 'myprog.byte'
```

> Passing the option `-tag "syntax(camlp4o)"` will also work in older versions. More generally, `-tag foo` will apply the tag `foo` to all targets, it is equivalent to adding `true: foo` in your tag line. Also note that the quoting, `-tag "syntax(camlp4o)"` instead of `-tag syntax(camlp4o)`, is necessary for your shell to understand tags that have parentheses.

If you use `-ppx` preprocessors, you can use the parametrized tag `ppx(…)` (`-tag "ppx(…)"`) to specify the preprocessor to use.

1.9. Archives, documentation

Some OCamlbuild features require you to add new kind of files in your source directory. Suppose you would like to distribute an archive file `mylib.cma` that would contain the compilation unit for your modules `mod1.ml` and `mod2.ml`. For this, you should create a file `mylib.mllib` listing the name of desired modules — capitalized, as in OCaml source code:

```
Mod1
Mod2
```

OCamlbuild knows about a rule `"%.mllib -> %.cma"`, so you can then use:

```
ocamlbuild 'mylib.cma'
```

or, for a native archive

```
ocamlbuild 'mylib.cmxa'
```

Producing a shared native library `.cmxs` is also supported by a different form of file with the same syntax, `foo.mldylib`.

Similarly, if you want to invoke `ocamldoc` to document your program, you should list the modules you want documented in a `.odocl` file. If you name it `mydoc.odocl` for example, you can then invoke:

```
ocamlbuild 'mydoc.docdir/index.html'
```

, which will produce the documentation in the subdirectory `mydoc.docdir`, thanks to a rule `"%.odocl → %.docdir/index.html"`.

1.10. Source and build directories, module paths, include paths

The "source directories" that `ocamlbuild` will traverse to look for rule dependencies are a subset of the subdirectory tree rooted at the "root directory", the place where you invoke `ocamlbuild`.

> All filesystem paths discussed in this section are **relative to the root directory**.

The build directory contains a copy of the source directories hierarchy, with the source file imported and additional targets produced during the previous builds. It is by convention the subdirectory `_build` of the root directory, although this can be set with the `-build-dir` command-line option. The build directory is not part of the source directories considered by OCamlbuild.

A subdirectory of the subdirectory tree is included in the source directories if it has the "traverse" tag set. That means that if you want to add "foo/bar" (and its files) as part of the source directories and remove "foo/baz", you can use the following in your `_tags` file:

```
"foo/bar": traverse
"foo/baz": -traverse
```

If the option `-r` (for *recursive*) is passed, then all subdirectories (recursively) are considered part of the source directories by default, except the build directory and directories that look like version-control information (`.svn`, `.bzr`, `.hg`, `.git`, `_darcs`).

This option is enabled by default *if* the root directory looks like an OCamlbuild project: either a `myocamlbuild.ml` or a `_tags` file is present.

> The reason for this heuristic is that calling `ocamlbuild` from your home directory could take a very long time if it recursively traverses your subdirectories, to check for hygiene for example.

If the root directory does not look like an OCamlbuild project, but you still wish to use it as such, you can just add the `-r` option explicitly. In the other direction, you can explicitly disable recursive traverse with `true: -traverse` in your `_tags` file.

1.10.1. Module paths and include directories

On many occasions OCamlbuild needs you to indicate *compilation units* (set of source and object files for a given OCaml module) located somewhere in the source directories. The syntax to do this is to use an OCaml module name (in particular, capitalized), prefixed by the relative path from the root directory. For example, `bar/baz/Foo` will work with the files `bar/baz/[fF]oo.{ml[i],cm*}`, depending on the compilation phase.

For convenience, it is possible to add a source directory to the set of *include paths*: paths that do not have to be explicitly prefixed to each module path. If `bar/` is in the include path, you can refer to `bar/baz/Foo` as just `baz/Foo`. To add `bar` to the include path, one can pass the `-I bar` option to ocamlbuild, or tag `"bar": include` in the `_tags` file.

> Some have come to see the use of the same syntax `-I foo` in `ocamlbuild` and in OCaml compilers as a mistake, because the underlying concepts are rather different. In particular, it is *not* the case that passing `-I foo` to `ocamlbuild` will transfer this command-line option to underlying compiler invocation. If `foo` is inside the source directories, this should not be needed, and if it is outside you are encouraged to rely on `ocamlfind` packages instead of absolute paths.

2. Reference documentation

In this chapter, we cover the built-in targets and tags provided by OCamlbuild. We will omit features that are deprecated, because we found they lead to bad practices or were superseded by better options. Of course, given that a `myocamlbuild.ml` can add new rules and tags, this documentation will always be incomplete.

2.1. File extensions of the OCaml compiler and common tools

A large part of the file extensions in OCamlbuild rules have not been designed by OCamlbuild itself, but are standard extensions manipulated by the OCaml compiler. As you may not be familiar with them, we will recapitulate them now. For most use-cases OCamlbuild will hide most of those subtleties from you, but having this reference is still useful to understand advanced usage scenarios or read build logs.

The OCaml compilers also accept native files (name extensions `{.o,.obj}`, `{.a,.lib}`) and even source files (`.c`) as input arguments, which get passed to the C toolchain (compiler or linker) when producing mixed C/OCaml programs or libraries.

Table 2. File name extensions

File name, with extension	Description
`foo.mll`	Lexer description, to be processed by a lexer generator to produce a `foo.ml` file, and possibly `foo.mli`.
`foo.ml`	Grammar description, to be processed by a parser generator to produce a `foo.ml` file, and possibly `foo.mli`.
`foo.cmo`	OCaml bytecode-compiled object file, providing the implementation of the module `Foo`.
`foo.cmi`	OCaml bytecode-compiled object file, providing the interface of the module `Foo`.
`blah.cma`	OCaml bytecode-compiled archive file, containing a collection of `.cmo` or `.cma` files, to be used as a library (for either static or dynamic linking).

File name, with extension	Description
`foo.cmx`	OCaml native-compiled object file, providing the implementation of the module `Foo`

`foo.o`
under Windows:
`foo.obj`
	Complementary native object file for the module `Foo`.

`foo.cmxa`	OCaml native-compiled archive file, containing a collection of `.cmx` files, for static linking only.
`foo.a` `foo.lib`	Complementary native library files for a native-compiled archive file `foo.cmxa`, containing a collection of `.o` or `.obj` files.
`foo.cmxs`	OCaml native-compiled archive file (or "plugin") for dynamic linking, containing a collection of `.cmx`, `.cmxa`, `.o|.obj`, `.a|.lib` files.

In addition, the following extensions are not enforced by the compiler itself, but are commonly used by OCaml tools:

Table 3. Additional, commonly found file name extensions

File name, with extension	Description
`foo.mll`	Lexer description, to be processed by a lexer generator to produce a `foo.ml` file, and possibly `foo.mli`.
`foo.mly`	Grammar description, to be processed by a parser generator to produce a `foo.ml` file, and possibly `foo.mli`.

`foo.mlp`
`foo.ml4`
`foo.mlip`
`foo.mli4`
	Common extensions for files to be processed by an external preprocessor (`p` for "preprocessing" and `4` for Camlp4, an influential OCaml preprocessor).

2.2. Targets and rules

The built-in OCamlbuild targets for OCaml compilation all rely on file extensions to know which rule to use. Note that this is not imposed by OCamlbuild rule system, which would allow more flexible patterns. But it is always the filename of the target that determines which rules to apply to build it.

In consequence, OCamlbuild adds specific file extensions to the one listed above (or variations of them), that are the user-interface to use its rules providing certain features. For example, `.inferred.mli` is not a standard extension in the OCaml compiler, but it is understood by a built-in rule of OCamlbuild to ask for the `.mli` that the compiler can auto-generate by typing a `.ml` file without an explicit interface: running `ocamlbuild foo.inferred.mli` will first build `foo.ml` (or find it in the source directory), then generate `foo.inferred.mli` from it. You are expected to then inspect `foo.inferred.mli`, ideally add documentation, and then move it to `foo.mli` by themselves.

The target extensions understood by OCamlbuild built-in rules are listed in the following subsections. Again, note that `myocamlbuild` plugins may add new targets and rules.

2.2.1. Basic targets

Table 4. Basic targets

File name extension	Description
`.cmi` `.cmo` `.cmx`	Builds those intermediate files from the corresponding source files (`.ml`, and the `.mli` if it exists).
`.byte` `.native`	Executables generated from a module and its dependencies for bytecode and native compilation.
`.mllib`	Contains a list of module paths (`Bar`, `subdir/Baz`) that will be compiled and archived together to build a corresponding `.cma` or `.cmxa` target.
`.cma` `.cmxa`	The preferred way to build a library archive is to use a `.mllib` file listing its content. If a `foo.mllib` is absent, building the target `foo.cm{,x}a` will create an archive with `foo.cm{o,x}` and all the local module it depends upon, transitively.
`.mldylib`	Contains a list of module paths (`Bar`, `subdir/Baz`) that will be compiled and archived together to build a corresponding `.cmxs` target (native plugin). There is no corresponding concept of bytecode plugin archive, as `.cma` files (built from `.mllib` files) support for static and dynamic linking.
`.cmxs`	The preferred way to build a plugin archive is to list its content in a `.mldylib` file. In absence of `foo.mldylib`, building `foo.cmxs` will either + * build `foo.cmxa` and copy its content into a `.cmxs` file (in particular this means that a `.cmxs` can be created from a `.mllib` file), or * build `foo.cmx` and create a plugin archive containing exactly `foo.cmx`. This differs from the rule for `.cm{,x}a` files (whose archive include the dependencies of the module `Foo`), in order to avoid dynamically linking the same modules several times.
`.itarget` `.otarget`	Building `foo.itarget` requests the build of the targets listed (one per line) in the corresponding `foo.itarget` file.

2.2.2. `ocamldoc` targets

These targets will call the documentation generator `ocamldoc`.

Table 5. `ocamldoc` targets

File name	Description
`.odocl`	Contains a list of module names for which to produce documentation, using one of the targets listed below.
`.docdir/index.html`	Building the target `foo.docdir/index.thml` will create a subdirectory `foo.docdir` containing the HTML documentation of all modules listed in `foo.odocl`.
`.docdir/man`	As `.docdir/index.html` above, but builds the documentation in `man` format.

File name	Description
`.docdir/bar.tex` or `.docdic/bar.ltx`	Building the target `foo.docdir/bar.tex` will build the documentation for the modules listed in `foo.odocl`, as a LaTeX file named `foo.docdir/bar.tex`. The basename `bar` is not important, but it is the extension `.tex` or `.ltx` that indicates to OCamlbuild that ocamldoc should be asked for a LaTeX output.
`.docdir/bar.texi`	Same as above, but generates documentation in TeXinfo format.
`.docdir/bar.dot`	Same as above, but generates a `.dot` graph of inter-module dependencies.

2.2.3. `ocamlyacc` and Menhir targets

OCamlbuild will by default use `ocamlyacc`, a legacy parser generator that is included in the OCaml distribution. The third-party parser generator Menhir is superior in all aspects, so you are encouraged to use it instead. To enable the use of Menhir instead of `ocamlyacc`, you should pass the `-use-menhir` option, or have `true: use_menhir` in your `_tags` file. OCamlbuild will then activate menhir-specific builtin rule listed below.

Table 6. `ocamlyacc` and Menhir targets

File name extension	Description
`.mly`	Grammar description files. They will be passed on to `ocamlyacc` to produce the corresponding `.ml` file, or Menhir if it is enabled.
`.mlypack`	Menhir (not ocamlyacc) supports building a parser by composing several `.mly` files together, containing different parts of the grammar description. Listing module paths in `foo.mlypack` will produce `foo.ml` and `foo.mli` by combining the `.mly` files corresponding to the listed modules.
`.mly.depends` `.mlypack.depends`	Menhir (but not `ocamlyacc`) supports calling `ocamldep` to approximate the dependencies of the OCaml module on which the generated parser will depend.

2.2.4. Advanced targets

Table 7. Advanced targets

File name extension	Description
`.ml.depends` `.mli.depends`	Call the `ocamldep` tool to compute a conservative over-approximation of the external dependencies of the corresponding source file.
`.inferred.mli`	Infer a `.mli` interface from the corresponding `.ml` file.

File name extension	Description
`.mlpack`	Contains a list of module paths (`Bar`, `subdir/Baz`) that can be packed as submodules of a `.cmo` or `.cmx` file: if `foo.mlpack` exist, asking for the target `foo.cmx` will build the modules listed in `foo.mlpack` and pack them together. [NOTE] .Packed submodules ==== The native OCaml compiler requires the submodules that will be packed to be compiled with the `-for-pack Foo` option (where `Foo` is the name of the result of packing), and OCamlbuild does not hide this semantics from the user: you can use the built-in parametrized flag `for-pack(Foo)` for this purpose. For example, to build `foo.cmx` containing `Bar` and `subdir/Baz` as packed-submodules, you should have the following: ---- foo.mlpack: Bar subdir/Baz _tags: <{bar,subdir/baz}.cmx: for-pack(Foo) ---- ====

`.byte.o`
`.byte.obj` *on Windows*
 	
`.byte.so`
`.byte.dll` *on Windows*
`.byte.dylib` *on OSX*
 	
`.byte.c`
	Produces object files for static or dynamic linking, or a C source file, by passing the `-output-obj` option to the OCaml bytecode compiler — see `-output-obj` documentation.

`.native.{o,obj}` `.native.{so,dll,dylib}`	Produces object files for static or dynamic linking by passing the `-output -obj` option to the OCaml native compiler — see `-output-obj` documentation.
`.c` `.{o,obj}`	OCamlbuild can build `.{o,obj}` files from `.c` files by passing them to the OCaml compiler (which in turns calls the C toolchain). The OCaml compiler called is `ocamlc` or `ocamlopt`, depending on whether or not the `native` flag is set on the `.c` source file.
`.clib`	Contains a list of file paths (e.g., `foo.o`, not module paths) to be linked together (by using the standard `ocamlmklib` tool) to produce a `.a` or `.lib` archive (for static linking) or a `.so` or `.dll` archive (for dynamic linking). The .clib name should be prefixed by `lib`, and the target name will then a `lib` or `dll` prefix, following standard conventions: to build a static library from `libfoo.clib`, you should require the target `libfoo.{a,lib}`, and to build a dynamic library you should require the target `dllfoo.{so,dll}`. If `foo.o` is listed and OCamlbuild is run from Windows, `foo.obj` will be used instead.
`.mltop` `.top`	Requesting the build of `foo.top` will look for a list of module paths in `foo.mltop`, and build a custom toplevel with all these modules pre-linked using the standard `ocamlmktop` tool.

2.3. Deprecated targets

Table 8. Deprecated targets

File name extension	Description
`.p.*` `.d.*`	OCamlbuild supports requesting `foo.p.{cmx,native}` and `foo.d.{cmo,byte}` to build libraries or executables with profiling information (`.p`) or debug information (`.d`) incorporated. Unfortunately, this runs counter the simple scheme used by the OCaml compiler to find the object files of a compilation unit dependencies: if `Foo` depends on a module `Bar`, the compilation of `foo.p.cmx` will inspect `bar.cmx` (rather than `bar.p.cmx`) for cross-module information — this is why `.d` is not supported for native code, as this defeats the purpose of debug builds. NOTE: `.p` is not supported for bytecode because bytecode profiling works very differently from native profiling. The more robust solution is to build `foo.{cmo,cmx,byte,native}` with the `profile` or `debug` flag set (e.g., `ocamlbuild -tag 'debug' 'foo.native'`, or using the `_tags` file). If the flag is set for certain files only, only those will have debugging or profiling information enabled. Note that (contrarily to the `.d.cmx` approach) this means you cannot keep a both a with-debug-info and a without-debug-info compiled object file for the same module at the same time: building `foo.byte` with `true: debug`, then without (or conversely) will rebuild all the `.cmo` files of all of `foo` dependencies each time.
`.pp.ml`	This target produces a pretty-printing (as OCaml source code) of the OCaml AST produced by preprocessing the corresponding `.ml` file. This does not work properly when using `ocamlfind` to activate Camlp4 preprocessors (the now-preferred way to enable syntax extensions), because `ocamlfind` does not provide a way to obtain the post-processing output, only to preprocess during compilation. Note that passing the `-dsource` compilation flag to the OCaml compiler will make it emit the result post-processing during compilation (as OCaml source code; use `-dparsetree` for a tree view of the AST).

2.4. Tags

2.4.1. Basic tags

For convenience, we try to offer a tag for each setting exported as command-line parameters by the OCaml compilers and tools. A builtin tag `foo_bar` corresponding to the option `-foo-bar` is in general better than trying to pass `-cflags -foo-bar` to the ocamlbuild compilation command, as it can enable the `-foo-bar` flag only when it make sense, in a more fine-grained way that "during a compilation command".

> If you notice that a compiler-provided command-line option is missing its tag counterpart, this is a bug that you should report against OCamlbuild. Feel free to look at the implementation and send a patch adding this tag, it is really easy.

Compiler tags

- `absname`
- `annot`
- `asm` (for `ocamlopt’s `-S`)

- `bin_annot`
- `compat_32`
- `custom`
- `debug` (for `-g`)
- `dtypes`
- `for-pack(PackModule)`
- `inline(5)`
- `keep_locs`
- `linkall`
- `no_alias_deps`
- `no_float_const_prop`
- `nolabels`
- `nopervasives`
- `opaque`
- `open(MyPervasives)`
- `output_obj`
- `output_shared` (for `-cclib -shared`, automatically set by `.{byte,native}.{so,dll,dylib}` targets)
- `pp(my_pp_preprocessor)`
- `ppx(my_ppx_preprocessor)`
- `principal`
- `profile` (for `-p`)
- `rectypes`
- `runtime_variant(_pic)`
- `safe_string`
- `short_paths`
- `strict_formats`
- `strict_sequence`
- `thread`
- `unsafe_string`
- `warn(A@10-28@40-42-45)`
- `warn_error(+10+40)`

`ocamlfind` tags

- `package(pkgname)`
- `linkpkg`
- `dontlink(pkgname)`
- `predicate(foo)`
- `syntax(bar)`

`ocamllex` tags

- `quiet` (`-q`)

- `generate_ml` (`-ml`)

Menhir tags

- `only_tokens`
- `infer`
- `explain`
- `external_tokens(TokenModule)`

`camlp4` **tags**

- `use_caml4_{,bin}`
- `camlp4{rrr,orrr,oof,orf,rf,of,r,o}{,.opt}`

2.4.2. Deprecated tags

The tags `use_{ocamlbuild,ocamldoc,toplevel,graphics,dbm,nums,bigarray,str,unix,dynlink}` were designed to indicate that the tagged modules depend on the corresponding libraries from the OCaml distributions (`use_{ocamlbuild,ocamldoc,toplevel}` allows to compile against the tools' libraries to build plugins). We now recommend to enable those libraries through their corresponding `ocamlfind` package.

2.4.3. `ocamlbuild`-specific tags

- `not_hygienic` and `precious`: explicitly indicate that a file is part of the source directory and should not be warned about by the hygiene-checking tools. This is useful if for some reason you are given, for example, a `.cmi` file to use unchanged in your project.
- `traverse`: explicitly indicate that OCamlbuild should consider this subdirectory as part of the current project; this flag is set for all subdirectories by default (so OCamlbuild will look in subdirectories recursively to find module dependencies) as soon as the current directory "looks like an OCamlbuild project" (there is either a `myocamlbuild.ml` or `tags file present). This tag is usefully used negative, "foo": -traverse, to say that a part of the local directory hierarchy should _not` be considered by OCamlbuild.
- `include`, `traverse`: see the section above on source directories and include paths.
- global tags: setting `true: use_mehir` in the root `_tags` file is equivalent to passing the `-use-menhir` command-line parameter.

2.5. Advanced (context) tags

These tags are generally not meant to be used directly in `_tags` file, but rather to serve as the context part of tag declarations. For example, the `link` flag is automatically added thet set of tags of linking-related command, allowing tag declarations to add specific flags during linking phase only — but it would make little sense to explicitly add the `link` tag to a target in your `_tags` file.

- language context: `c` or `ocaml` indicate whether the compiler invocation are working with OCaml files, or C files (to be passed to the underlying C toolchain). If you wished to use OCamlbuild for a completely different purpose (not necessarily OCaml-related), for example building LaTeX documents, you could use a corresponding `latex` tag.

- compilation stage context: `pp` (syntactic preprocessing), `compile` (compilation of source files), `link` (linking of object files), but also `pack` (when packing compiled object files), `library` (when creating library archives), `infer_interface` (producing a `.mli` from the corresponding `.ml`) and `toplevel` (when building custom toplevels).
- byte or native compilation context: `byte` (`ocamlc`) or `native` (`ocamlopt`).
- extension tags: when building the target `foo.bar`, a tag `extension:bar` is added to the set of current tags. This is used by the builtin `ocamldoc` rules to enable either `-latex` or `-dot` depending on the requested target extension.
- tool-specific tags: `menhir`, `ocamlyacc`, `ocamllex`, `doc` (for `ocamldoc`)

2.6. The `-documentation` option

Invoking `ocamlbuild -documentation` will give a list of rules and tags known to OCamlbuild in the current project (including those defined in the `myocamlbuild` plugin). This is a good way to quickly look for the tag name corresponding to a particular option, and also more accurate than the above reference manual, which does cannot describe plugin-specific features.

> This output is sensitive to the current configuration. For example, `ocamlbuild -use -ocamlfind -documentation` and `ocamlbuild -no-ocamlfind -documentation` produce different outputs, as the latter does not include `ocamlfind`-specific tags.

Here is an example of rule documentation included in the `ocamlbuild -documentation` output:

```
rule "ocaml: modular menhir (mlypack)"
  ~deps:[ %.mlypack ]
  ~prods:[ %.mli; %.ml ]
  ~doc:"Menhir supports building a parser by composing several .mly files
        together, containing different parts of the grammar description. To
        use that feature with ocamlbuild, you should create a .mlypack file
        with the same syntax as .mllib or .mlpack files: a
        whitespace-separated list of the capitalized module names of the .mly
        files you want to combine together."
  your_build_function
```

> The previous sample resembles OCaml code (see Rule declarations), but is not valid OCaml code.

Note that rule declaration only indicate the static dependencies of rules (those that determine whether or not the rule will be tried). This rule is explicit about the fact that invoking Menhir produces both a `.ml` and `.mli`.

2.6.1. Parameterized tags

Parameterized tags are documented since OCaml version 4.03. An example:

```
parametrized flag {. compile, ocaml, ppx(my_ppx) .} "-ppx my_ppx"

[...]

flag {. compile, no_alias_deps, ocaml .} "-no-alias-deps"
```

2.7. Syntax of _tags files

A _tags file associates file name patterns with tags like this:

```
# This is a comment
true: bin_annot, debug
<protocol_*> or <main.*>: package(yojson)
```

The general syntax is:

```
{pattern}: {comma-separated tag list}
```

These items are ended by a newline. Comments (starting with #) and empty lines are ignored, and an escaped line break is considered as whitespace (so those items can span multiple lines).

The {pattern} part is what we call a "glob expression", which is an expression built of basic logic connective on top of "glob patterns".

Table 9. The syntax of glob expressions

Syntax	Example	Meaning
<p>	<foo.*>	Paths matching the pattern p.
"s"	"foo/bar.ml"	The exact string s.
e1 or e2	<*.ml> or <foo/bar.ml>	Paths matching at least one of the expression e1 or e2.
e1 and e2	<*.ml> and <foo_*>	Paths matching both expressions e1 and e2.
not e	not <*.mli>	Paths not matching the expression e.
true	true	All pathnames.
false	false	Nothing.
(e)	(<*> and not <*.*>)	Same as e (useful for composing larger expressions).

Table 10. The syntax of glob patterns

Syntax	Example	Matches	Does not match	Meaning
s	foo.ml	foo.ml	bar.ml	The exact string s.

Syntax	Example	Matches	Does not match	Meaning

`*`
(wildcard)
	`*`	The empty path
		`foo`
		`bar`
			`foo/bar`
			`/baz`
				Any string not containing a slash `/`.

`?`
(joker)
	`?`	`a`
		`b`
		`z`
			`/`
			`/bar`
				Any one-letter string, excluding the slash `/`.

`**/`
(prefix inter-directory wildcard)
	`**/foo.ml`	`foo.ml`
		`bar/foo.ml`
		`bar/baz/foo.ml`
			`bar/foo`	The empty string, or any string ending with a slash `/`.

`/**` *(suffix inter-directory wildcard)*	`foo/**`	`foo` `foo/bar`	`bar/foo`	The empty string, or any string starting with a slash `/`.
`/**/` *(infix inter-directory wildcard)*	`bar/**/foo.ml`	`bar/foo.ml` `bar/baz/foo.ml`	`foo.ml`	Any string starting and ending iwth a slash `/`.

`[r1 r2 r3 …]`,
where each `r` is either:
* A single character `c`; or * A range `c1-c2`
(positive character class).
	`[a-fA-F0-9_.]`	`3`
		`F`
		`.`
			`z`
			`bar`
				Any one-letter string made up of characters from one of the given ranges.

`[^ r1 r2 r3 …]`,
where each `r` is either: * A single character `c`; or * A range `c1-c2`
(negative character class).
	`[a-fA-F0-9_.]`	`z`
		`bar`
			`3`
			`F`
			`.`
				Any one-letter string **not** made up of characters from one of the given ranges.

`p1 p2`
(concatenation)
	`foo*`	`foo`
		`foob`
		`foobar`
			`fo`
			`bar`
				Any string with a (possibly empty) prefix matching the pattern `p1` and the (possibly empty) remainder matching the pattern `p2`.

`{ p1, p2, … }` *(union)*	`toto.{ml,mli}`	`toto.ml` `toto.mli`	`toto.`	Any string matching one of the given patterns.

In addition, rule patterns may include pattern variables. `%(foo: p)` will match for the pattern `p` and name the result `%(foo)`. For example, `%(path: <**/>)foo.ml` is useful. `%(foo)` will match the pattern `true` and name the result `%(foo)`, and finally `%` will match the pattern `true` and match the result `%`. Consider the following examples:

```
%.cmx
%(dir).docdir/%(file)
%(path:<**/>)lib%(libname:<*> and not <*.*>).so
```

3. Enriching OCamlbuild through plugins

3.1. How `myocamlbuild.ml` works

If you have a `myocamlbuild.ml` file at the root of your OCamlbuild project, the building process will run in two steps.

3.1.1. First step

First, OCamlbuild will compile `myocamlbuild.ml`, linking it with all the modules that are part of the globally installed `ocamlbuild` executable. This will produce a program `_build/myocamlbuild` that behaves exactly like `ocamlbuild` itself, except that it also runs the code of your `myocamlbuild.ml` file. Immediately after, OCamlbuild will stop (before doing any work on the targets you gave it) and start the `_build/myocamlbuild` program instead, that will handle the rest of the job. This is quite close to how, for example, XMonad (a window manager whose configuration files are pure Haskell) works.

This means that it is technically possible to do anything in `myocamlbuild.ml` that could be done by adding more code to the upstream OCamlbuild sources. But in practice, relying on the implementation internals would be fragile with respect to OCamlbuild version changes.

We thus isolated a subset of the OCamlbuild API, exposed by the `Ocamlbuild_plugin` module, that defines a stable interface for plugin developers. It lets you manipulate command-line options, define new rules and targets, add new tags or refine the meaning of existing flags, etc. The signature of this module is the `PLUGIN` module type of the interface-only `signatures.mli` file of the OCamlbuild distribution.

> You will find the module source code littered with comments explaining the purpose of the exposed values, but this documentation aspect can still be improved. We warmly welcome patches to improve this aspect of `ocamlbuild` — or any other aspect.

You can influence the `myocamlbuild.ml` compilation-and-launch process in several ways:

- The `no-plugin` option allows to ignore the `myocamlbuild.ml` file and just run the stock `ocamlbuild` executable on your project. This means that fancy new rules introduced by `myocamlbuild.ml` will not be available.
- The `-just-plugin` option instructs OCamlbuild to stop compilation after having built the plugin. It also guarantees that OCamlbuild will try to compile the plugin, which it may not always do, for example when you only ask for cleaning or documentation.
- The `-plugin-option FOO` option will pass the command-line option `FOO` to the `myocamlbuild` invocation — and ignore it during plugin compilation.

- The `-plugin-tag` and `-plugin-tags` options allow to pass tags that will be used to compile the plugin. For example, if someone develops a nice library to help writing OCamlbuild plugins and distribute as 'toto.ocamlbuild' in `ocamlfind` then `-plugin-tag "package(toto.ocamlbuild)"` will let you use it in your `myocamlbuild.ml`.

> The rationale for `-plugin-option` and `-plugin-tag` to apply during different phases of the process is that an option is meaningful at runtime for the plugin, while a plugin tag is meaningful at compile-time.

3.1.2. Dispatch

Tag and rule declarations, or configuration option manipulation, are side-effects that modify a global OCamlbuild state. It would be fragile to write your `myocamlbuild.ml` with such side-effects performed at module initialization time, in the following style:

```
open Ocamlbuild_plugin
(* bad style *)
let () =
  Options.ocamlc := "/better/path/to/ocamlc"
;;
```

The problem is that you have little idea, and absolutely no flexibility, of the time at which those actions will be performed with respect to all the other actions of OCamlbuild. In this example, command-line argument parsing will happen after this plugin effect, so the changed option would be overridden by command-line options, which may or may not be what a plugin developer expects.

To alleviate this side-effect order issue, OCamlbuild lets you register actions at hook points, to be called at a well-defined place during the OCamlbuild process. If you want your configuration change to happen after options have been processed, you should in fact write:

```
open Ocamlbuild_plugin
let () =
  dispatch (function
    | After_options ->
      Options.ocamlc := "..."
    | _ -> ())
```

The `dispatch` function register a hook-listening function provided by the user. Its type is `(hook → unit) → unit`. The hooks are currently defined as:

```
(** Here is the list of hooks that the dispatch function have to handle.
    Generally one responds to one or two hooks (like After_rules) and do
    nothing in the default case. *)
type hook =
  | Before_hygiene
  | After_hygiene
  | Before_options
  | After_options
  | Before_rules
  | After_rules
```

> OCamlbuild does not guarantee any order in which it will call various hooks, except of course that `Before_foo` always happens before `After_foo`. In particular, the `hygiene` hooks may be called before or after other hooks, or not be called at all if OCamlbuild decides not to check Hygiene.

3.1.3. Flag declarations

A flag declaration maps a *set of tags* to a list of command-line arguments/flags/options/parameters. These arguments will be added to a given compilation command if each of the tags are present on the given target.

The following example can be found in `ocaml_specific.ml`, the file of the OCamlbuild sources that defines most OCaml-specific tags and rules of OCamlbuild:

```
flag ["ocaml"; "annot"; "compile"] (A "-annot");
```

This means that the `-annot` command-line option is added to any compilation command for which those three tags are present. The tags `"ocaml"` and `"compile"` are activated by default by OCamlbuild, `"ocaml"` for any ocaml-related command, and `"compile"` specifically for compilation steps — as opposed to linking, documentation generation, etc. The `"annot"` flag is not passed by default, so this tag declaration will only take effects on targets that are explicitly marked `annot` in the `_tags` file.

This very simple declarative language, mapping sets of tags to command-line options, is the way to give meaning to OCamlbuild tags — either add new ones or overload existing ones. It is very easy, for example, to pass a different command-line argument depending on whether byte or native-compilation is happening.

```
flag ["ocaml"; "use_camlp4_bin"; "link"; "byte"]
  (A"+camlp4/Camlp4Bin.cmo");
flag ["ocaml"; "use_camlp4_bin"; "link"; "native"]
  (A"+camlp4/Camlp4Bin.cmx");
```

The `A` constructor stands for ‘atom(ic)’, and is part of a `spec` datatype, representing specifications of fragments of command. We will not describe its most advanced constructors — it is again exposed and documented in `signatures.mli` — but the most relevant here are:

```
(** The type for command specifications. That is pieces of command. *)
and spec =
  | N              (** No operation. *)
  | S of spec list (** A sequence.  This gets flattened in the last stages *)
  | A of string    (** An atom. *)
  | P of pathname  (** A pathname. *)
  [...]
```

> When introducing new flags, it is sometime difficult to guess which combination of tags to use. A hint to find the right combination is to have a look at OCamlbuild's log file that is saved in `_build/_log` each time `ocamlbuild` is run. It contains the targets OCamlbuild tried to produce, with the associated list of tags and the corresponding command lines.

Parametrized tags

You can also define families of parametrized tags such as `package(foo)` or `inline(30)`. This is done through the `pflag` function, which takes a list of usual tags, the special parametrized tag, and a function from the tag parameter to the corresponding command specification. Again from the `PLUGIN` module type in `signatures.mli`:

```
(** Allows to use [flag] with a parametrized tag (as [pdep] for [dep]).

    Example:
       pflag ["ocaml"; "compile"] "inline"
         (fun count -> S [A "-inline"; A count])
    says that command line option "-inline 42" should be added
    when compiling OCaml modules tagged with "inline(42)". *)
val pflag : Tags.elt list -> Tags.elt -> (string -> Command.spec) -> unit
```

Rule declarations

OCamlbuild let you build your own rules, to teach it how to build new kind of targets. This is done by calling the `rule` function from a plugin, which is declared and documented in the `PLUGIN` module in `signatures.mli`. We will not write an exhaustive documentation here (for this, have a look at `signatures.mli`), but rather expose the most common features through representative examples.

Our first example is simple, as it is a rule without dynamic dependencies:

```
rule "ocaml dependencies ml"
  ~prod:"%.ml.depends"
  ~dep:"%.ml"
  ~doc:"call ocamldep to compute a syntactic over-approximation \\
        of the dependencies of the corresponding implementation file"
  ocamldep_ml_command
```

The first string parameter is the name of the rule. This rule tells OCamlbuild how to build `foo.ml.depends` from its `foo.ml`. The `%` character here is a pattern variable: if the target name (e.g., `foo.ml.depends`) matches the pattern of the rule production `%.ml.depends` then OCamlbuild will try to build the static dependency of the rule. The static dependency is the evaluation of `%.ml` in the pattern-matching environment `% → "foo"` (that is, `foo.ml`). If this static dependency can be built, then the "action" `ocamldep_ml_command` will be invoked to produce the expected result.

The action, of type `PLUGIN.action`, is a function that takes the current pattern-matching environment (in our example, mapping the pattern variable `%` to `foo`), a builder function, and returns a command, the command to execute to produce the final target of this rule. A plugin developer defining this rule should define the `ocamldep_ml_command` as follows:

```
let ocamldep_ml_command env _build =
  let arg = env "%.ml" and out = env "%.ml.depends" in
  let tags = tags_of_pathname arg ++ "ocaml" ++ "ocamldep" in
  Cmd(S[A "ocamldep"; T tags; A "-modules"; P arg; Sh ">"; Px out])
```

The first line in this definition uses the pattern environment to compute the actual name of the input and output files. These are then passed in as arguments to the `ocamldep` command and shell redirect, respectively, on the third line. The environment type `PLUGIN.env` is just `string → string`, it takes a pattern and substitutes its pattern variables to return a closed result.

The second line in this definition computes the tags to include in the command invocation. When OCamlbuild is passed back the command, it uses the tag declarations to determine which, if any, additional flags to insert into the command invocation. The call `tags_of_pathname arg` looks up in the `_tags` file any tags associated with file `foo.ml`. To these tags the rule code also adds the two contextual tags `ocaml` and `ocamldep` (on which flag declarations may depend).

Finally, the command is built:

```
Cmd(S[A "ocamldep"; T tags; A "-modules"; P arg; Sh ">"; Px out])
```

We already mentioned above the constructors `S`, `A` and `P` of the `command.spec` type:

- `S` just builds a sequence by concatenating sequent fragments;
- `A` is used for 'atoms' (fragments of text to be included as-is, but may be escaped to make them syntactic shell code); and
- `P` denotes a filesystem path that should be quoted.

The constructor `T` is used to embed tags within a command. Mind that passing `T` twice, one with the tag set `ocaml, ocamldep` and the other with the tag `foo`, is not equivalent to passing `ocaml, ocamldep, foo` together, as the transformation of tags into flags proceeds on each `T` fragment separately.

`Sh` is used for bits of raw shell code that should not be quoted at all, here the output redirection `>`. Finally, `Px` indicates a filesystem path just as `P`, but it adds the information that this filesystem path is the path of the target produced by this rule. This information is used by OCamlbuild for logging purposes.

> *Tags handling in rules*
>
> It is entirely the rule author's responsibility to include tags in the action's command. In particular, it is the code of the rule's action that decides which, if any, tags are taken into account and if they come from the rule dependencies, products or both. (Unfortunately, the built-in rules themselves are sometimes inconsistent on this.)

Dynamic dependencies

In the action `ocamldep_ml_command` of the previous example, the `~build` parameter (of type `PLUGIN.builder`) was ignored, because the rule had no dynamic dependencies. Therefore there was no need to build extra targets determined during the execution of the rule itself. The static dependency is built by `ocamlbuild`'s resolution engine before the action executed.

For example:

```
let target_list env build =
    let itarget = env "%.itarget" in
    let targets =
      let dir = Pathname.dirname itarget in
      let files = string_list_of_file itarget in
      List.map (fun file -> [Pathname.concat dir file]) files
    in
    let results = List.map Outcome.good (build targets) in
    let link_command result =
      Cmd (S [A "ln"; A "-sf";
              P (Pathname.concat !Options.build_dir result);
              A Pathname.pwd])
    in
    Seq (List.map link_command results)

rule "target files"
  ~dep:"%.itarget"
  ~stamp:"%.otarget"
  ~doc:"If foo.itarget contains a list of ocamlbuild targets, \
        asking ocamlbuild to produce foo.otarget will \
        build each of those targets in turn."
  target_list
```

The `string_list_of_file` function reads a file and returns the list of its lines. It is used in the various built-in rules for files containing other file or module paths (e.g., `.mllib`, `.odocl` or here `.itarget`).

The function `build` takes as argument a list of lists, to be understood as a conjunction of disjunctions. For example, if passed the input `[["a/foo.byte"; "b/foo.byte"]; ["a/foo.native"; "b/foo.native"]]`, it tries to build ((`a/foo.byte` OR `b/foo.byte`) AND (`a/foo.native` OR `b/foo.native`)). The disjunctive structure (this OR that) is useful because we are often not quite sure where a particular target may be (for example the module `Foo` may be in any of the subdirectories in the include path). The conjunctive structure (this AND that) is essential to parallelizing the build, since `ocamlbuild` tries to build all these targets in parallel, whereas sequential invocation of the build function on each of the disjunctions would give sequential builds.

The function `build` returns a list of outcomes (`(string, exn) Outcome.t` — `Outcome.t` is just a disjoint-sum type), that is either a `string` (the possible target that could be built) or an exception. `Outcome.good` returns the good result if it exists, or raises the exception.

3.2. Stamps

In the rule above, the production `"%.otarget"` is not passed as `~prod` parameter, but as a `~stamp`. Stamps are special files that record the list of digests of the dynamic dependencies of the rule that produced them.

This is useful to know whether a target should be re-compiled, or whether it is already up-to-date from a previous build and can be just kept as-is. Imagine that a rule to produce a file `foo.weird` depends on the rules listed in the corresponding `foo.itarget` (and then performs some build action). When should `foo.weird` be rebuilt, and when is it up-to-date? More precisely, after we have built the targets of `foo.itarget`, how do we know whether we should re-run the build action of `foo.weird`? Obviously, just checking if the `foo.itarget` file changed is not enough (the list of targets could be identical and yet, if one of the target changed, `foo.weird` must be rebuilt).

This is where `foo.otarget` comes in: because it contains a list of digests of the dependencies of `foo.itarget`, `foo.weird` can statically depend on `foo.otarget`. `foo.weird` does not need to depend on `foo.itarget` directly, and it will transitively depend on it through `foo.otarget`. This stamp file will change each time one of the `foo.itarget` elements changes, and thus `foo.weird` will be rebuilt exactly as necessary.

Such stamps should be used each time a rule has no natural file output to use as output (the case of `.itarget`), or when this file output does not contain enough information for its digest to correctly require rebuilding. The latter case occurs in the rule to build `ocamldoc` documentation `%.docdir/index.html`: the `index.html` only lists the documented modules. It does not contain their documentation — that is in other files generated. The rule thus produces a stamp `%.docdir/html.stamp`. You should only depend on that stamp if you want your rule to be executed *each time* the documentation changes.

3.3. Pattern variables

Most rules need exactly one pattern variable and use `%` for this purpose. You may use any string of the form `%(identifier)` as pattern variable, or even `%(identifier:pattern)`, in which case the pattern will be only be matched by a string matching the corresponding `pattern`. For example, the rule to produce the dynamic library archive `dllfoo.so` from the file list `libfoo.clib` starts as follows:

```
rule "ocaml C stubs: clib & (o|obj)* -> (a|lib) & (so|dll)"
    ~prods:(["%(path:<**/>)lib%(libname:<*> and not <*.*>)"-.-ext_lib] @
          if Ocamlbuild_config.supports_shared_libraries then
            ["%(path:<**/>)dll%(libname:<*> and not <*.*>)"-.-ext_dll]
          else
            [])
    ~dep:"%(path)lib%(libname).clib"
```

3.3.1. Complete example: Menhir support in OCamlbuild

```
rule "ocaml: modular menhir (mlypack)"
  ~prods:["%.mli" ; "%.ml"]
  ~deps:["%.mlypack"]
  ~doc:"Menhir supports building a parser by composing several .mly files \
        together, containing different parts of the grammar description. \
        To use that feature with ocamlbuild, you should create a .mlypack \
        file with the same syntax as .mllib or .mlpack files: \
        a whitespace-separated list of the capitalized module names \
        of the .mly files you want to combine together."
  (Ocaml_tools.menhir_modular "%" "%.mlypack" "%.mlypack.depends");

rule "ocaml: menhir modular dependencies"
  ~prod:"%.mlypack.depends"
  ~dep:"%.mlypack"
  (Ocaml_tools.menhir_modular_ocamldep_command "%.mlypack" "%.mlypack.depends");

rule "ocaml: menhir"
  ~prods:["%.ml"; "%.mli"]
  ~deps:["%.mly"; "%.mly.depends"]
  ~doc:"Invokes menhir to build the .ml and .mli files derived from a .mly \
        grammar. If you want to use ocamlyacc instead, you must disable the \
        -use-menhir option that was passed to ocamlbuild."
  (Ocaml_tools.menhir "%.mly");

rule "ocaml: menhir dependencies"
  ~prod:"%.mly.depends"
  ~dep:"%.mly"
  (Ocaml_tools.menhir_ocamldep_command "%.mly" "%.mly.depends");

flag ["ocaml"; "menhir"] (atomize !Options.ocaml_yaccflags);

flag [ "ocaml" ; "menhir" ; "explain" ] (S[A "--explain"]);
flag [ "ocaml" ; "menhir" ; "infer" ] (S[A "--infer"]);

List.iter begin fun mode ->
  flag [ mode; "only_tokens" ] (S[A "--only-tokens"]);
  pflag [ mode ] "external_tokens" (fun name ->
    S[A "--external-tokens"; A name]);
        end [ "menhir"; "menhir_ocamldep" ];
```

, where the Menhir-specific actions in `Ocaml_tools` are defined as follows:

```
let menhir_ocamldep_command' tags ~menhir_spec out =
  let menhir = if !Options.ocamlyacc = N then V"MENHIR" else !Options.ocamlyacc in
  Cmd(S[menhir; T tags; A"--raw-depend";
        A"--ocamldep"; Quote (ocamldep_command' Tags.empty);
        menhir_spec ; Sh ">"; Px out])
```

```
let menhir_ocamldep_command arg out env _build =
  let arg = env arg and out = env out in
  let tags = tags_of_pathname arg++"ocaml"++"menhir_ocamldep" in
  menhir_ocamldep_command' tags ~menhir_spec:(P arg) out

let import_mlypack build mlypack =
  let tags1 = tags_of_pathname mlypack in
  let files = string_list_of_file mlypack in
  let include_dirs = Pathname.include_dirs_of (Pathname.dirname mlypack) in
  let files_alternatives =
    List.map begin fun module_name ->
      expand_module include_dirs module_name ["mly"]
    end files
  in
  let files = List.map Outcome.good (build files_alternatives) in
  let tags2 =
    List.fold_right
      (fun file -> Tags.union (tags_of_pathname file))
      files tags1
  in
  (tags2, files)

let menhir_modular_ocamldep_command mlypack out env build =
  let mlypack = env mlypack and out = env out in
  let (tags,files) = import_mlypack build mlypack in
  let tags = tags++"ocaml"++"menhir_ocamldep" in
  let menhir_base = Pathname.remove_extensions mlypack in
  let menhir_spec = S[A "--base" ; P menhir_base ; atomize_paths files] in
  menhir_ocamldep_command' tags ~menhir_spec out

let menhir_modular menhir_base mlypack mlypack_depends env build =
  let menhir = if !Options.ocamlyacc = N then "menhir" else !Options.ocamlyacc in
  let menhir_base = env menhir_base in
  let mlypack = env mlypack in
  let mlypack_depends = env mlypack_depends in
  let (tags,files) = import_mlypack build mlypack in
  let () = List.iter Outcome.ignore_good (build [[mlypack_depends]]) in
  Ocaml_compiler.prepare_compile build mlypack;
  let ocamlc_tags = tags++"ocaml"++"byte"++"compile" in
  let tags = tags++"ocaml"++"parser"++"menhir" in
  Cmd(S[menhir ;
        A "--ocamlc"; Quote(S[!Options.ocamlc; T ocamlc_tags; ocaml_include_flags
mlypack]);
        T tags ; A "--base" ; Px menhir_base ; atomize_paths files])
```