The `ocamlbuild` users manual

Berke DURAK, Nicolas POUILLARD

February 2007

Abstract

`ocamlbuild` is a tool automating the compilation of most OCaml projects with minimal user input. Its use is not restricted to projects having a simple structure – the extra effort needed to make it work with the more complex projects is in reasonable proportion with their added complexity. In practice, one will use a set of small text files, and, if needed, an OCaml compilation module that can fine-tune the behaviour and define custom rules.

1 Features of `ocamlbuild`

This section is intended to read like a sales brochure or a datasheet.

- Built-in compilation rules for OCaml projects handle all the nasty cases: native and byte-code, missing `.mli` files, preprocessor rules, debugging and profiling flags, C stubs.
- Plugin mechanism for writing compilation rules and actions in a real programming language, OCaml itself.
- Automatic inference of dependencies.
- Correct handling of dynamically discovered dependencies.
- Object files and other temporary files are created in a specific directory, leaving your main directory uncluttered.
- Sanity checks ensure that object files are where they are supposed to be: in the build directory.
- Regular projects are built using a single command with no extra files.
- Parallel compilation to speed up things on multi-core systems.
- Sophisticated display mode to keep your screen free of boring and repetitive compilation message while giving you important progress information in a glimpse, and correctly multiplexing the error messages.
- Tags and flags provide a concise and convenient mechanism for automatic selection of compilation, preprocessing and other options.
- Extended shell-like glob patterns, that can be combined using boolean operators, allow you to concisely define the tags that apply to a given file.
- Mechanisms for defining the mutual visibility of subdirectories.
- Cache mechanism avoiding unnecessary compilations where reasonably computable.

2 Limitations

Not perfect nor complete yet, but already pretty damn useful.

We were not expecting to write the ultimate compilation tool in a few man-months, however we believe we have a tool that solves many compilation problems, especially our own, in a satisfactory way. Hence there are a lot of missing features, incomplete options and hideous bugs lurking in `ocamlbuild`, and we hope that the OCaml community will find our first try at `ocamlbuild` useful and hopefully help it grow into a tool that satisfies most needs of most users by providing feedback, bug reports and patches.

The plugin API maybe somewhat lacking in maturity, as it has only been tested by a few people. We believe a good API can only evolve under pressure from many peers and the courage to rewrite things cleanly when time is ripe by the developers. Most of the important functions a user will need are encapsulated in the plugin API, which is the `Ocamlbuild_plugin` module pack. We intend to keep that API backwards compatible. It may happen that intricate projects need features not available in that module – you may then use functions or values directly from the core `ocamlbuild` modules. We ask you to report such usage to the authors so that we may make the necessary changes to the API; you may also want to isolate

calls to the non-API parts of the `ocamlbuild` library from the rest of your plugin to to be able to keep the later when incompatible changes arise.

The way that `ocamlbuild` handles the command-line options, the `_tags` file, the target names, names of the tags, and so on, are not expected to change in incompatible ways. We intend to keep a project that compiles without a plugin compilable without modifications in the future.

3 Using `ocamlbuild`

Learn how to use `ocamlbuild` with short, specific, straight-to-the-point examples.

The amount of time and effort spent on the compilation process of a project should be proportionate to that spent on the project itself. It should be easy to set up a small project, maybe a little harder for a medium-sized project, and it may take some more time, but not too much, for a big project. Ideally setting up a big project would be as easy as setting up a small project. However, as projects grow, modularization techniques start to be used, and the probability of using meta programming or multiple programming languages increases, thus making the compilation process more delicate.

`ocamlbuild` is intended to be very easy to use for projects, large or small, with a simple compilation process: typing `ocamlbuild foo.native` should be enough to compile the native version of a program whose top module is `foo.ml` and whose dependencies are in the same directory. As your project gets more complex, you will gradually start to use command-line options to specify libraries to link with, then configuration files, ultimately culminating in a custom OCaml plugin for complex projects with arbitrary dependencies and actions.

3.1 Hygiene & where is my code ?

Your code is in the `_build` directory, but `ocamlbuild` automatically creates a symbolic link to the executables it produces in the current directory. `ocamlbuild` copies the source files and compiles them in a separate directory which is `_build` by default.

For `ocamlbuild`, any file that is not in the build directory is a source file. It is not unreasonable to think that some users may have bought binary object files they keep in their project directory. Usually binary files cluttering the project directory are due to previous builds using other systems. `ocamlbuild` has so-called "hygiene" rules that state that object files (`.cmo`, `.cmi`, or `.o` files, for instance) must not appear outside of the build directory. These rules are enforced at startup; any violations will be reported and `ocamlbuild` will exit. You must then remove these files by hand or run, with caution, the script `sterilize.sh`, which is generated in your source directory. This script will contain commands to remove them for you.

To disable these checks, you can use the `-no-hygiene` flag. If you have files that must elude the hygiene squad, just tag them with `precious` or `not_hygienic`.

3.2 Hello, world !

Assuming we are in a directory named `example1` containing one file `hello.ml` whose contents are

```
let _ =
  Printf.printf "Hello, %s ! My name is %s\n"
    (if Array.length Sys.argv > 1 then Sys.argv.(1) else "stranger")
    Sys.argv.(0)
;;
```

we can compile and link it into a native executable by invoking `ocamlbuild hello.native`. Here, `hello` is the base-name of the top-level module and `native` is an extension used by `ocamlbuild` to denote native code executables.

```
% ls
hello.ml
% ocamlbuild hello.native
Finished, 4 targets (0 cached) in 00:00:00.
% ls -l
total 12
drwxrwx--- 2 linus gallium 4096 2007-01-17 16:24 _build/
-rw-rw---- 1 linus gallium   43 2007-01-17 16:23 hello.ml
lrwxrwxrwx 1 linus gallium   19 2007-01-17 16:24 hello.native -> _build/hello.native*
-rw-r----- 1 linus gallium  460 2007-01-17 16:24 _log
```

What's this funny `_build` directory ? Well that's where `ocamlbuild` does its dirty work of compiling. You usually won't have to look very often into this directory. Source files are be copied into `_build` and this is where the compilers will be run. Various cache files are also stored there. Its contents may look like this:

```
% ls -l _build
total 208
-rw-rw---- 1 linus gallium    337 2007-01-17 16:24 _digests
-rw-rw---- 1 linus gallium    191 2007-01-17 16:24 hello.cmi
-rw-rw---- 1 linus gallium    262 2007-01-17 16:24 hello.cmo
-rw-rw---- 1 linus gallium    225 2007-01-17 16:24 hello.cmx
-rw-rw---- 1 linus gallium     43 2007-01-17 16:23 hello.ml
-rw-rw---- 1 linus gallium     17 2007-01-17 16:24 hello.ml.depends
-rwxrwx--- 1 linus gallium 173528 2007-01-17 16:24 hello.native*
-rw-rw---- 1 linus gallium    936 2007-01-17 16:24 hello.o
-rw-rw---- 1 linus gallium     22 2007-01-17 16:24 ocamlc.where
```

3.3 Executing my code

You can execute your code the old-fashioned way (`./hello.native`). You may also type

```
ocamlbuild main.native -- Caesar
```

and it will compile and then run `main.native` with the arguments following `--`, which should display:

```
% ocamlbuild hello.native -- Caesar
Finished, 4 targets (0 cached) in 00:00:00.
Hello, Caesar ! My name is _build/hello.native
```

3.4 The log file, verbosity and debugging

By default, if you run `ocamlbuild` on a terminal, it will use some ANSI escape sequences to display a nice, one-line progress indicator. To see what commands `ocamlbuild` has actually run, you can check the contents of the `_log` file. To change the name of the log file or to disable logging, use the `-log <file>` or `-no-log` options. Note that the log file is truncated at each execution of `ocamlbuild`.

The log file contains all the external commands that `ocamlbuild` ran or intended to run along with the target name and the computed tags. With the `-verbose <level>` option, `ocamlbuild` will also write more or less useful debugging information; a verbosity level of 1 (which can also be specified using the `-verbose` switch) prints generally useful information; higher levels produce much more output.

3.5 Cleaning

`ocamlbuild` may leave a `_build` directory, symbolic links to executables in that directory, and a `_log` file. All of these can be removed safely by hand, or by invoking `ocamlbuild` with the `-clean` flag.

3.6 Where and how to run `ocamlbuild`?

An important point is that `ocamlbuild` must be invoked from the root of the project, even if this project has multiple, nested subdirectories. This is because `ocamlbuild` likes to store the object files in a single `_build` directory. You can change the name of that directory with the `-build-dir` option.

`ocamlbuild` can be either invoked manually from the UNIX or Windows shell, or automatically from a build script or a Makefile. Unless run with the `-no-hygiene` option, there is the possibility that `ocamlbuild` will prompt the user for a response. By default, on UNIX systems, if `ocamlbuild` senses that the standard output is a terminal, it will use a nice progress indicator using ANSI codes, instrumenting the output of the processes it spawns to have a consistent display. Under non-UNIX systems, or if the standard output is not a terminal, it will run in classic mode where it will echo the executed commands on its standard output. This selection can be overridden with the `-classic-display` option.

3.7 Dependencies

Dependencies are automatically discovered.

Most of the value of `ocamlbuild` lies in the fact that it often needs no extra information to compile a project besides the name of the top-level module. `ocamlbuild` calls `ocamldep` to automatically find the dependencies of any modules it wants to compile. These dependencies are dynamically incorporated in the dependency graph, something `make` cannot do. For instance, let's add a module `Greet` that implements various ways of greeting people.

```
% cat greet.ml
type how = Nicely | Badly;;

let greet how who =
```

```
  match how with Nicely -> Printf.printf "Hello, %s !\n" who
                | Badly  -> Printf.printf "Oh, here is that %s again.\n" who
;;
% cat hello.ml
open Greet

let _ =
  let name =
    if Array.length Sys.argv > 1 then
      Sys.argv.(1)
    else
      "stranger"
  in
  greet
    (if name = "Caesar" then Nicely else Badly)
    name;
  Printf.printf "My name is %s\n" Sys.argv.(0)
;;
```

Then the module `Main` depends on the module `Greet` and `ocamlbuild` can figure this out for himself – we still only have to invoke `ocamlbuild main.native`. Needless to say, this works for any number of modules.

3.8 Native and byte-code

If we want to compile byte-code instead of native, we just a target name of `main.byte` instead of `main.native`, i.e., we type `ocamlbuild hello.byte`.

3.9 Compile flags

To pass a flag to the compiler, such as the `-rectypes` option, use the `-cflag` option as in:

```
ocamlbuild -cflag -rectypes hello.native
```

You can put multiple `-cflag` options, they will be passed to the compiler in the same order. You can also given them in a comma-separated list with the `-cflags` option (notice the plural):

```
ocamlbuild -cflags -I,+lablgtk,-rectypes hello.native
```

These flags apply when compiling, that is, when producing `.cmi`, `.cmo`,`.cmx` and `.o` files from `.ml` or `.mli` files.

3.10 Link flags

Link flags apply when the various object files are collected and linked into one executable. These will typically be include directories for libraries. They are given using the `-lflag` and `-lflags` options, which work in the same way as the `-cflag` and `-cflags` options.

3.11 Linking with external libraries

In our third example, we use one Unix system call and functions from the `num` library:

```
% cat epoch.ml
let _ =
  let s = Num.num_of_string (Printf.sprintf "%.0f" (Unix.gettimeofday ())) in
  let ps = Num.mult_num (Num.num_of_string "1000000000000") s in
  Printf.printf "%s picoseconds have passed since January 1st, 1970.\n"
    (Num.string_of_num ps)
;;
```

This requires linking with the `unix` and `num` modules, which is accomplished by using the `-lib unix` and `-lib num` flags, or, alternatively, `-libs unix,num`:

```
% ocamlbuild -libs nums,unix epoch.native --
Finished, 4 targets (4 cached) in 00:00:00.
116905164700000000000 picoseconds have passed since January 1st, 1970.
```

You may need to add options such as `-cflags -I,/usr/local/lib/ocaml/` and `-lflags -I,/usr/local/lib/ocaml/` if the libraries you wish to link with are not in OCaml's default search path.

3.12 The `_tags` files

Finer control over the compiler flags applied to each source file, such as preprocessing, debugging, profiling and linking options, can be gained using `ocamlbuild`'s tagging mechanism.

Every source file has a set of tags which tells `ocamlbuild` what kind of file it is and what to do with it. A tag is simply a string, usually lowercase, for example `ocaml` or `native`. The set of tags attached to a file is computed by applying the tagging rules to the filename. Tagging rules are defined in `_tags` files in any parent directory of a file, up to the main project directory.

Each line in the `_tags` file is made of a glob pattern (see subsection 3.13) and a list of tags. More than one rule can apply to a file and rules are applied in the order in which they appear in a file. By preceding a tag with a minus sign, one may remove tags from one or more files.

3.12.1 Example: the built-in `_tags` file

```
<**/*.ml> or <**/*.mli> or <**/*.mlpack> or <**/*.ml.depends>: ocaml
<**/*.byte>: ocaml, byte, program
<**/*.odoc>: ocaml, doc
<**/*.native>: ocaml, native, program
<**/*.cma>: ocaml, byte, library
<**/*.cmxa>: ocaml, native, library
<**/*.cmo>: ocaml, byte
<**/*.cmi>: ocaml, byte, native
<**/*.cmx>: ocaml, native
```

A special tag made from the path name of the file relative to the toplevel of the project is automatically defined for each file. For a file `foo/bar.ml` this tag will be `file:foo/bar.ml`.

If you do not have subdirectories, you can put `*.ml` instead of `**/*.ml`.

3.13 Glob patterns and expressions

Glob patterns have a syntax similar to those used by UNIX shells to select path names (like `foo_*.ba?`). They are used in `ocamlbuild` to define the files and directories to which tags apply. Glob expressions are glob patterns enclosed in brackets < and > combined using the standard boolean operators `and`, `or`, `not`. This allows one to describe sets of path names in more concise and more readable ways.

Please note that file and directory names are supposed to be made of the following characters: `a`, ..., `z`, `A`, ..., `Z`, `0`, ..., `9`, `_`, `-` and `.`. This is called the pathname alphabet P.

3.14 Subdirectories

If the files of your project are held in one or more subdirectories, `ocamlbuild` must be made aware of that fact using the `-I` or `-Is` options or by adding an `include` tag. For instance, assume your project is made of three subdirectories, `foo`, `bar` and `baz` containing various `.ml` files, the main file being `foo/main.ml`. Then you can either type:

```
% ocamlbuild -Is foo,bar,baz foo/main.native
```

or add the following line in the `_tags` file

```
<foo> or <bar> or <baz>: include
```

and call

```
% ocamlbuild foo/main.native
```

There are then two cases. If no other modules named `Bar` or `Baz` exist elsewhere in the project, then you are done. Just use `Foo`, `Foo.Bar` and `Foo.Baz` in your code. Otherwise, you will need to use the plugin mechanism and define the mutual visibility of the subdirectories using the XXX function.

3.15 Grouping targets with `.itarget`

You can create a file named `foo.itarget` containing a list of targets, one per line, such as

```
main.native
main.byte
stuff.docdir/index.html
```

Requesting the target `foo.otarget` will then build every target listed in the file `foo.itarget`. Blank lines and dashes to comment out lines are accepted.

Formal syntax	*Example*	*Matches*	*Does not match*	*Meaning (formal meaning)*
u A string of pathname characters	`foo.ml`	`foo.ml`	`fo.ml`, `bar/foo.ml`	The exact string u ($\{u\}$, where $u \in P^*$)
`*` The wild-card star	`*`	ε, `foo`, `bar`	`foo/bar`, `/bar`	Any string not containing a slash (P^*)
`?` The joker	`?`	`a`, `b`, `z`	`/`, `bar`	Any one-letter string, excluding the slash
`**/` The prefix inter-directory star	`**/foo.ml`	`foo.ml`, `bar/foo.ml`, `bar/baz/foo.ml`	`foo/bar`, `/bar`	The empty string, or any string ending with a slash ($\varepsilon \cup P^*/$)
`/**` The suffix inter-directory star	`foo/**`	`foo`, `foo/bar`	`bar/foo`	Any string starting with a slash, or the empty string. ($\varepsilon \cup /P^*$)
`/**/` The infix inter-directory star	`bar/**/foo.ml`	`bar/foo.ml`, `bar/baz/foo.ml`	`foo.ml`	Any string starting and ending with a slash ($\varepsilon \cup /P^*/$)
$[r_1 r_2 \cdots r_k]$ where r_i is either c or $c_1 - c_2$ ($1 \leq i \leq k$) The positive character class	`[a-fA-F0-9_.]`	`3`, `F`, `.`	`z`, `bar`	Any one-letter string made of characters from one of the ranges r_i ($1 \leq i \leq n$). ($\mathscr{L}(r_1) \cup \cdots \cup \mathscr{L}(r_n)$)
`[^`$r_1 r_2 \cdots r_k$`]` where r_i is either c or $c_1 - c_2$ ($1 \leq i \leq k$) The negative character class	`[^a-fA-F0-9_.]`	`z`, `bar`	`3`, `F`, `.`	Any one-letter string NOT made of characters from one of the ranges r_i ($1 \leq i \leq n$). ($\Sigma^* \setminus (\mathscr{L}(r_1) \cup \cdots \cup \mathscr{L}(r_n))$)
$p_1 p_2$ A concatenation of patterns	`foo*`	`foo`, `foob`, `foobar`	`fo`, `bar`	Any string with a prefix matching p_1 and the corresponding suffix matching p_2, ($\{uv \mid u \in \mathscr{L}(p_1), v \in \mathscr{L}(p_2)\}$)
$\{p_1, p_2, \cdots, p_k\}$ A union of patterns	`toto.{ml,mli}`	`toto.ml`, `toto.mli`	`toto.`	Any string matching one of the patterns p_i for $1 \leq i \leq k$. ($\mathscr{L}(p_1) \cup \cdots \cup \mathscr{L}(p_k)$)

Table 1: Syntax and semantics of glob patterns.

Formal syntax	*Example*	*Meaning (formal meaning)*
<p>	`<foo.ml>`	Pathnames matching the pattern p
e_1 **or** e_2	`<*.ml> or <foo/bar.ml>`	Pathnames matching at least one of the expressions e_1 and e_2
e_1 **and** e_2	`<*.ml> and <foo_*>`	Pathnames matching both expressions e_1 and e_2
not e	`not <*.mli>`	Pathnames not matching the expression e
true	`true`	All pathnames
false	`false`	No pathnames

Table 2: Syntax and semantics of glob expressions.

3.16 Packing subdirectories into modules

OCaml's `-pack` option allows you to structure the contents of a module in a subdirectory. For instance, assume you have a directory `foo` containing two modules `bar.ml` and `baz.ml`. You want from these to build a module `Foo` containing `Bar` and `Baz` as submodules. In the case where no modules named `Bar` or `Baz` exist outside of `Foo`, To do this you must write a file `foo.mlpack`, preferably sitting in the same directory as the directory `Foo` and containing the list of modules (one per line) it must contain:

```
Bar
Baz
```

3.17 Preprocessor options and tags

You can specify preprocessor options with `-pp` followed by the preprocessor string, for instance `ocamlbuild -pp "camlp4o.op -unsafe"` would run your sources thru CamlP4 with the `-unsafe` option. Another way is to use the tags file.

Tag	Display character
ocaml	O
native	N
byte	B
program	P
pp	R
debug	D
interf	I
link	L

Table 3: Relation between the characters displayed in the tag string and the tags.

Tag	Preprocessor command	Remark
`pp(cmd...)`	`cmd...`	Arbitrary preprocessor command[1]
`camlp4o`	`camlp4o`	Original OCaml syntax
`camlp4r`	`camlp4r`	Revised OCaml syntax
`camlp4of`	`camlp4of`	Original OCaml syntax with extensions
`camlp4rf`	`camlp4rf`	Revised OCaml syntax with extensions

3.18 Debugging byte code and profiling native code

The preferred way of compiling code suitable for debugging with `ocamldebug` or profiling native code with `ocamlprof` is to use the appropriate target extensions, `.d.byte` for debugging or `.p.native`.

Another way is to add use the `debug` or `profile` tags. Note that these tags must be applied at the compilation and linking stages. Hence you must either use `-tag debug` or `-tag profile` on the command line, or add a

```
true: debug
```

line to your `_tags` file. Please note that the byte-code profiler works in a wholly different way and is not supported by `ocamlbuild`.

3.19 Generating documentation using `ocamldoc`

Write the names of the modules whose interfaces will be documented in a file whose extension is `.odocl`, for example `foo.odocl`, then invoke `ocamlbuild` on the target `foo.docdir/index.html`. This will collect all the documentation from the interfaces (which will be build, if necessary) using `ocamldoc` and generate a set of HTML files under the directory `foo.docdir/`, which is actually a link to `_build/foo.docdir/`. As for packing subdirectories into modules, the module names must be written one per line, without extensions and correctly capitalized. Note that generating documentation in formats other than HTML or from implementations is not supported.

3.20 The display line

Provided `ocamlbuild` runs in a terminal under a POSIX environment, it will display a sophisticated progress-indicator line that graciously interacts with the output of subcommands. This line looks like this:

```
00:00:02 210   (180 ) main.cmx                                ONbp--il /
```

Here, 00:00:02 is the elapsed time in hour:minute:second format since `ocamlbuild` has been invoked; 210 is the number of external commands, typically calls to the compiler or the like, that may or may not have been invoked; 180 is the number of external commands that have not been invoked since their result is already in the build directory; `main.cmx` is the name of the last target built; `ONbp-il` is a short string that describes the tags that have been encountered and the slash at the end is a frame from a rotating ticker. Hence, the display line has the following structure:

```
HH:MM:SS JOBS (CACHED) PATHNAME                                TAGS TICKER
```

The tag string is made of 8 indicators which each monitor a tag. These tags are `ocaml`, `native`, `byte`, `program`, `pp`, `debug`, `interf` and `link`. Initially, each indicator displays a dash `-`. If the current target has the monitored tag, then the indicator displays the corresponding character (see table 3) in uppercase. Otherwise, it displays that character in lowercase. This allows you to see the set of tags that have been applied to files in your project during the current invocation of `ocamlbuild`.

Hence the tag string `ONbp-il` means that the current target `main.cmx` has the tags `ocaml` and `native`, and that the tags `ocaml`, `native`, `byte`, `program`, `interf` and `link` have already been seen.

3.21 `ocamllex`, `ocamlyacc` and `menhir`

`ocamlbuild` knows how to run the standard lexer and parser generator tools `ocamllex` and `ocamlyacc` when your files have the standard `.mll` and `.mly` extensions. If you want to use `menhir` instead of `ocamlyacc`, you can either launch `ocamlbuild` with the `-use-menhir` option or add a

```
true: use_menhir
```

line to your `_tags` file. Note that there is currently no way of using `menhir` and `ocamlyacc` in the same execution of `ocamlbuild`.

3.22 Changing the compilers or tools

As `ocamlbuild` is part of your OCaml distribution, it knows if it can call the native compilers and tools (`ocamlc.opt`, `ocamlopt.opt`...) or not. However you may want `ocamlbuild` to use another `ocaml` compiler for different reasons (such as cross-compiling or using a wrapper such as `ocamlfind`). Here is the list of relevant options:

- `-ocamlc <command>`
- `-ocamlopt <command>`
- `-ocamldep <command>`
- `-ocamlyacc <command>`
- `-menhir <command>`
- `-ocamllex <command>`
- `-ocamlmktop <command>`
- `-ocamlrun <command>`

3.23 Writing a `myocamlbuild.ml` plugin

3.24 Interaction with version control systems

Here are tips for configuring your version control system to ignore the files and directories generated by `ocamlbuild`.

The directory `_build`, the file `_log` and any symbolic links pointing into `_build` should be ignored. To do this, you must add the following ignore patterns to your version control system's ignore set:

```
_log
_build
*.native
*.byte
*.d.native
*.p.byte
```

For CVS, add the above lines to the `.cvsignore` file. For Subversion (SVN), type `svn propedit svn:ignore .` and add the above lines.

3.25 A shell script for driving it all?

To shell or to make ? Traditionally, makefiles have two major functions. The first one is the dependency-ordering, rule-matching logic used for compiling. The second one is as a dispatcher for various actions defined using phony targets with shell script actions. These actions include cleaning, cleaning really well, archiving, uploading and so on. Their characteristic is that they rely little or not on the building process – they either need the building to have been completed, or they don't need anything. As `/bin/sh` scripts have been here for three to four decades and are not going anywhere, why not replace that functionality of makefiles with a shell script ? We have thought of three bad reasons:

- Typing `make` to compile is now an automatism,
- We need to share variable definitions between rules and actions,
- Escaping already way too special-character-sensitive shell code with invisible tabs and backslashes is a dangerously fun game.

We also have bad reasons for not using an OCaml script to drive everything:

- `Sys.command` calls the `/bin/sh` anyway,

- Shell scripts can execute partial commands or commands with badly formed arguments.
- Shell scripts are more concise for expressing... shell scripts.

Anyway you are of course free to use a makefile or an OCaml script to call ocamlbuild. Here is an example shell driver script:

```
#!/bin/sh

set -e

TARGET=epoch
FLAGS="-libs unix,nums"
OCAMLBUILD=ocamlbuild

ocb()
{
  $OCAMLBUILD $FLAGS $*
}

rule() {
  case $1 in
    clean)  ocb -clean;;
    native) ocb $TARGET.native;;
    byte)   ocb $TARGET.byte;;
    all)    ocb $TARGET.native $TARGET.byte;;
    depend) echo "Not needed.";;
    *)      echo "Unknown action $1";;
  esac;
}

if [ $# -eq 0 ]; then
  rule all
else
  while [ $# -gt 0 ]; do
    rule $1;
    shift
  done
fi
```

A Motivations

This inflammatory appendix describes the frustration that led us to write `ocamlbuild`.

Many people have painfully found that the utilities of the `make` family, namely GNU Make, BSD Make, and their derivatives, fail to scale to large projects, especially when using multi-stage compilation rules, such as custom pre-processors, unless dependencies are hand-defined. But as your project gets larger, more modular, and uses more diverse pre-processing tools, it becomes increasingly difficult to correctly define dependencies by hand. Hence people tend to use language-specific tools that attempt to extract dependencies. However another problem then appears: `make` was designed with the idea of a static dependency graph. Dependency extracting tools, however, are typically run by a rule in `make` itself; this means that make has to reload the dependency information. This is the origin of the `make clean; make depend; make` mantra. This approach tends to work quite well as long as all the files sit in a single directory and there is only one stage of pre-processing. If there are two or more stages, then dependency extracting tools must be run two or more times - and this means multiple invocations of `make`. Also, if one distributes the modules of a large project into multiple subdirectories, it becomes difficult to distribute the makefiles themselves, because the language of `make` was not conceived to be modular; the only two mechanisms permitted, inclusion of makefile fragments, and invocation of other make instances, must be skillfully coordinated with phony target names (`depend1, depend2...`) to insure inclusion of generated dependencies with multi-stage programming; changes in the structure of the project must be reflected by hand and the order of variable definitions must be well-thought ahead to avoid long afternoons spent combinatorially fiddling makefiles until it works but no one understands why.

These problems become especially apparent with OCaml: to ensure type safety and to allow a small amount of cross-unit optimization when compiling native code, interface and object files include cryptographical digests of interfaces they are to be linked with. This means that linking is safer, but that makefile sloppiness leads to messages such as:

```
Files foo.cmo and bar.cmo
make inconsistent assumptions over interface Bar
```

The typical reaction is then to issue the mantra `make clean; make depend; make` and everything compiles just fine... from the beginning. Hence on medium projects, the programmer often has to wait for minutes instead of the few seconds that would be taken if `make` could correctly guess the small number of files that really had to be recompiled.

It is not surprising that hacking a build tool such as `make` to include a programming language while retaining the original syntax and semantics gives an improvised and cumbersome macro language of dubious expressive power. For example, using GNU make, suppose you have a list of `.ml`s that you want to convert into a list including both `.cmo`s and `.cmi`s, that is you want to transform `a.ml b.ml c.ml` into `a.cmi a.cmo b.cmi b.cmo c.cmi c.cmo` while preserving the dependency order which must be hand specified for linking [2]. Unfortunately `$patsubst %.ml, %.cmi %.cmo, a.ml b.ml c.ml` won't work since the %-sign in the right-hand of a `patsubst` gets substituted only once. You then have to delve into something that is hardly lambda calculus: an intricate network of `foreach`, `eval`, `call` and `define`s may get you the job done, unless you chicken out and opt for an external `awk`, `sed` or `perl` call. People who at this point have not lost their temper or sanity usually resort to metaprogramming by writing Makefile generators using a mixture of shell and m4. One such an attempt gave something that is the nightmare of wannabe package maintainers: it's called `autotools`.

Note that it is also difficult to write `Makefiles` to build object files in a separate directory. It is not impossible since the language of `make` is Turing-complete, a proof of which is left as an exercise. Note that building things in a separate directory is not necessarily a young enthusiast's way of giving a different look and feel to his projects – it may be a good way of telling the computer that `foo.mli` is generated by `ocamlyacc` using `foo.mly` and can thus be removed.

B Default rules

Tags	Dependencies	Targets
	%.itarget	%.otarget
ocaml	%.mli %.mli.depends	%.cmi
byte, debug, ocaml	%.mlpack %.cmi	%.d.cmo
byte, ocaml	%.mlpack	%.cmo %.cmi
byte, ocaml	%.mli %.ml %.ml.depends %.cmi	%.d.cmo
byte, ocaml	%.mli %.ml %.ml.depends %.cmi	%.cmo
native, ocaml, profile	%.mlpack %.cmi	%.p.cmx %.p.o
native, ocaml	%.mlpack %.cmi	%.cmx %.o
native, ocaml, profile	%.ml %.ml.depends %.cmi	%.p.cmx %.p.o
native, ocaml	%.ml %.ml.depends %.cmi	%.cmx %.o
debug, ocaml	%.ml %.ml.depends %.cmi	%.d.cmo
ocaml	%.ml %.ml.depends	%.cmo %.cmi
byte, debug, ocaml, program	%.d.cmo	%.d.byte
byte, ocaml, program	%.cmo	%.byte
native, ocaml, profile, program	%.p.cmx %.p.o	%.p.native
native, ocaml, program	%.cmx %.o	%.native
byte, debug, library, ocaml	%.mllib	%.d.cma
byte, library, ocaml	%.mllib	%.cma
byte, debug, library, ocaml	%.d.cmo	%.d.cma
byte, library, ocaml	%.cmo	%.cma
	lib%(libname).clib	lib%(libname).a dll%(libname).so
	%(path)/lib%(libname).clib	%(path)/lib%(libname).a %(path)/dll%(libname).so
library, native, ocaml, profile	%.mllib	%.p.cmxa %.p.a
library, native, ocaml	%.mllib	%.cmxa %.a
library, native, ocaml, profile	%.p.cmx %.p.o	%.p.cmxa %.p.a
library, native, ocaml	%.cmx %.o	%.cmxa %.a
	%.ml	%.ml.depends
	%.mli	%.mli.depends
ocaml	%.mll	%.ml
doc, ocaml	%.mli %.mli.depends	%.odoc
	%.odocl	%.docdir/index.html
ocaml	%.mly	%.ml %.mli
	%.c	%.o
	%.ml %.ml.depends	%.inferred.mli

[2] By the way, what's the point of having a declarative language if `make` can't sort the dependencies in topological order for giving them to `gcc` or whatever ?