Cheat sheet – OCaml tools and commands

Objectives

This document is a simple cheatsheet about useful commands and tools of the OCaml ecosystem.

In this document, `$>` denotes the prompt, meaning that a line beginning by `$>` is a command you can type on your GNU/Linux system. However, `$>` does not has to be typed, only the characters (*i.e.* the command that follows it). For instance:

```
$> ls
```

means that it is the command `ls` (list directory contents).

1 Tools and commands

1.1 opam

`opam` is the package manager of OCaml. It allows developers to download, install and compile libraries for OCaml (e.g. `utop`). Software are installed in `~./opam`.

The first command to type to create `~/.opam` is the following:

```
$> opam init
```

After that, the following command is necessary to have the access to the environment:

```
$> `opam config env`
```

Note #1: in the previous command line, the symbols are *backquotes* (or *backticks*), not quotes (`'`) nor any other symbol.

Note #2: it is necessary to type that command every time you open a terminal. Therefore, it can be smart to include that clause in your `.profile.perso` file (or `.bashrc`, `.bash_profile`, etc. depending on your environment).

1.2 ocaml

`ocaml` is the standard interactive interperter (REPL[1]) that reads expressions, evaluates them and prints the result. All the expressions have to finish by `;;`

`ocaml` can simply be launched by typing:

```
$> ocaml
```

1.3 utop

`utop`[2] is an user-friendly interactive interpreter that reads expressions, evaluates them and prints the result. It does the same work as `ocaml` but in a more user-friendly way. It is the REPL we use in this UE.

To install `utop` using `opam` on a computer, you can do it by using the following command:

```
$> opam install utop
```

[1] Read-Eval-Print-Loop

[2] `https://github.com/diml/utop`

`utop` can be launched by typing:
`$> utop`

`utop` prompt is denoted by a sharp `#`. `utop` *directives* also begin with a sharp (e.g. `#quit;;`).

1.4 ocamlc & ocamlrun

`ocamlc` (or `ocamlopt`) is the OCaml compiler. Let suppose OCaml code is in `mycode.ml` file. The following command:

`$> ocamlc -c myfile.ml`

will compile the signature file (`.mli`) if it exists, and compile the module.

The following command:

`$> ocamlc -o myprogram ...`

will link the modules to produce a *bytecode* file that is executable by the OCaml virtual machine (`ocamlrun`). This file can be executed by:

`$> ocamlrun myprogram`

or directly by:

`$> ./myprogram`

1.5 ocamllex

`ocamllex` is the tool we use to construct lexical analyzers.

A description of `ocamllex` is available here: `http://caml.inria.fr/pub/docs/manual-ocaml/lexyacc.html`

`ocamllex` takes a lexer written in a `.mll` file as input and produces the corresponding OCaml code (encoding an automaton). Let suppose the lexer is written in `myExpressionLexer.mll`, `ocamllex` can be used as following:

`$> ocamllex myExpressionLexer.mll`

The produced OCaml code can then be integrated in an other program (for instance as part of a compiler...).

1.6 menhir

`menhir` is the generator of syntactical analysers ("parser generator") we use in this UE. The documentation can be found here: `http://gallium.inria.fr/~fpottier/menhir/menhir.html` [en] or `http://gallium.inria.fr/~fpottier/menhir/menhir.html.fr` [fr].

`menhir` is installed by default on the school computers, but if you want to install it on your own computer, use the following command:

`$> opam install menhir`

`menhir` takes a set of rules written in a `.mly` file as input and produces the corresponding OCaml code. Let suppose the parser is written in `myExpressionParser.mly`, `menhir` can be used as following:

`$> menhir myExpressionParlser.mly`

The produced OCaml code can then be integrated in an other program (for instance as part of a compiler...).

1.7 ocamlbuild

`ocamlbuild` is a powerful tool that automates your OCaml builds. You should *really* use it whenever you develop *more-than-five-lines-of-code-ocaml-programs*.

There exists two kind of targets depending on the compiler one wants to use `ocamlc` (`.byte`) or `ocamlopt` (`.native`). For example:

```
$> ocamlbuild main.native
```

compiles the file `main.ml` and all its dependencies with `ocamlopt`. It will also link the program with the unix library and will produce an executable named `main.native`. Lastly, it will create a symbolic link in the current directory to the produce executable.

```
$> ocamlbuild main.byte
```

To run a built executable, add `--` followed by arguments:

```
$> ocamlbuild myprogram.byte -- inputfile.txt
```

For more information about `ocamlbuild`, please refer to the documentation of the tool: `https://github.com/ocaml/ocamlbuild/blob/master/manual/manual.adoc`

ocamlfind & _tags: If you want to reuse an existing OCaml library. Start by installing it with `opam`. For example, to use colored terminal output:

```
$> opam install ANSITerminal
```

Then you must inform `ocamlbuild` to use the `ocamlfind` tool:

```
$> ocamlbuild -use-ocamlfind Main.byte -- SomeInputFile.txt
```

and you must complete you `_tags` file with the library name. For instance, with the previous example, the `_tags` file will contain the following line:

```
true: package(ANSITerminal)
```