Ocaml Cheatsheet

CSCI 599, Fall 2020: An Introduction to Programming Languages

Mukund Raghothaman

August 27, 2020

This document has two purposes: (*a*) to familiarize me with the peculiarities of Ocaml syntax, and (*b*) to present *you*, the students of CSCI 599, with an operational overview of the Ocaml language, and to be a one-stop guide to some selected issues relating to its syntax and semantics, the students of CSCI 599. Notably, this document is neither intended to be a pedagogical introduction, nor a comprehensive language reference. The official language reference may be accessed at `https://caml.inria.fr/pub/docs/manual-ocaml/`.

Note: This document will be updated as the semester progresses, and is expected to be perpetually incomplete. Please check back regularly.

1 Installation

We summarize the instructions listed at `https://dev.realworldocaml.org/install.html`.

1. Install Opam using the instructions listed at `https://opam.ocaml.org/doc/Install.html`. The commands for some popular operating systems are as follows:
   a) On recent versions of Fedora, run `dnf install opam`.
   b) On recent versions of Ubuntu, run `add-apt-repository ppa:avsm/ppa; apt update; apt install opam`.
   c) On computers running OSX with the Homebrew package manager, run `brew install gpatch; brew install opam`.

   Prefix the `sudo` qualifier to the above commands as appropriate.

2. Initialize Opam and create a switch for this course:

```
$ opam init
$ eval $(opam env)
$ opam switch create fa2020-csci599 ocaml-base-compiler.4.10.0
$ eval $(opam env)
```

Confirm that the switch has been correctly initialized by running `opam switch`. Opam is a package manager for Ocaml packages.

3. Install the appropriate packages within the switch:

   ```
   $ opam install base core utop
   ```

   Place the following toplevel directives in your `.ocamlinit` file:

   ```
   #use "topfind";;
   #thread;;  (* Threads need to be enabled for Core to work *)
   #require "core.top";;
   ```

   We will dissect some of these commands in Section 2.

4. You may optionally install Ocaml language support on your editor. If you are using Visual Studio Code, this consists of the following two steps:

   a) Install the Ocaml Language Server Protocol (LSP):

      ```
      $ opam pin add ocaml-lsp-server \
                     https://github.com/ocaml/ocaml-lsp.git
      $ opam install ocaml-lsp-server
      ```

      The Language Server Protocol is a recent standard developed by IDE vendors to provide language-specific features within the IDE which can be best implemented by support from the compiler, debugger, static analyzer, or other parts of the framework.

   b) Install the IDE extension to provide language support. Either run

      ```
      $ code --install-extension ocamllabs.ocaml-platform
      ```

      from the command line, or navigate to File ▷ Preferences ▷ Extensions, and install the extension named "Ocaml Platform". You can find a more detailed description of this extension at https://marketplace.visualstudio.com/items?itemName=ocamllabs.ocaml-platform.

2 Interactive Development Using the REPL and Utop

1. On Semicolons: `;` and `;;`. See https://baturin.org/docs/ocaml-faq/.
   a) The expression `e1 ; e2` is equivalent to `let _ = e1 in e2`.
   b) The double-semicolon, `stmt;;` is an end-of-input marker for the top-level interpreter. Instead write either `let _ = stmt`, or `let () = stmt` to require that `stmt` has `unit` type.
2. The `.ocamlinit` File
3. Directives: `#use`, `#require`
4. `open`

3 The Bytecode Compiler, Native Code Compiler, and Build System

Two versions of the Ocaml compiler exist, which are guaranteed to be semantically equivalent: (*a*) the bytecode compiler, `ocamlc`, and (*b*) the native code compiler, `ocamlopt`. Both compilers produce an executable `a.out` file from a file containing Ocaml source code. However, the executable produced by `ocamlopt` is an optimized native code binary (in ELF format, if running on Linux):

```
$ ocamlopt test.ml
$ ldd a.out
        linux-vdso.so.1 (0x00007ffd95af0000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0
           x00007f8d76219000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0
           x00007f8d76213000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0
           x00007f8d76022000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f8d763ea000)
$ ocamlc test.ml
$ ldd a.out
        not a dynamic executable
```

4 Basic Data Types and Built-In Operations

See https://caml.inria.fr/pub/docs/manual-ocaml/expr.html#ss%3Aexpr-operators for the list of infix operations initially defined in the system, and https://caml.inria.fr/pub/docs/fpcl/fpcl-04.pdf for additional background. There are six basic data types:

1. The type of machine integers, `int`, as in `0`, `1`, `-4`, etc. Represented in 2-s complement form and supports the following operations.

   a) Infix arithmetic operations: addition, subtraction, multiplication, division, and integer modulus. Respectively, `(+)`, `(-)`, `(*)`, `(/)`, `(mod) : int -> int -> int`.

   b) Infix bitwise operations, `(land)`, `(lor)`, `(lxor)`, `(lsl)`, `(lsr)`, `(asr): int -> int -> int`. Prefix bitwise negation (i.e., 1-s complement), `lnot : int -> int`.

   c) The representational limits, `max_int`, `min_int : int`

   For example,

```
# 2 + 4;;
- : int = 6
# 8 mod 3;;
- : int = 2
# 3 land 5;;
- : int = 1
# max_int;;
- : int = 4611686018427387903
# min_int;;
- : int = -4611686018427387904
# lnot max_int = min_int;;
- : bool = true
```

2. The type of floating point numbers, `float`, as in `2.3`, `-5.8`, and `(2.)`. Represented in accordance with the IEEE-754 standard. As with integers, they support the following operations.

   a) Arithmetic: `(+.)`, `(-.)`, `(*.)`, `(/.)`, `(**) : float -> float -> float`. The expression x `**` y performs floating point exponentiation, x^y.

   b) Conversion to and from machine integers. Respectively, `int_of_float : float -> int` and `float_of_int : int -> float`.

   For example,

```
# 2.3 +. 4.8;;
- : float = 7.1
# -2.3;;
- : float = -2.3
# -.2.3;;
- : float = -2.3
# 2. ** 3.2;;
```

```
- : float = 9.18958683997628
# max_float;;
- : float = 1.7976931348623157e+308
# min_float;;
- : float = 2.2250738585072014e-308
# float_of_int 2;;
- : float = 2.
# int_of_float 3.2;;
- : int = 3
# int_of_float (-3.2);;
- : int = -3
```

3. The Boolean type, `bool`, of values `true` and `false`.
   a) Boolean conjunctions and disjunctions, `(&&), (||): bool -> bool -> bool`. The responses to a StackOverflow question (https://stackoverflow.com/q/23833221) indicate that short-circuiting evaluation is followed. Two variants, `(&), (or): bool -> bool -> bool`, have been marked as deprecated.
   b) Negation, `not : bool -> bool`.
   c) Structural equality and inequality, `(=), (<>): 'a -> 'a -> bool`. These are often the equality operators what you want.
   d) Physical equality and inequality, `(==), (!=): 'a -> 'a -> bool`.
   e) Ordinal comparisons, `(<), (<=), (>), (>=): 'a -> 'a -> bool`. Note the polymorphic comparison operators.

   Unsurprisingly,

```
# 2 = 3;;
- : bool = false
# 2 = 2;
- : bool = true
# 2. < 2.3;
- : bool = true
# "abc" < "abcd";;
- : bool = true
# "abd" > "ac";;
- : bool = false
# true && false;;
- : bool = false
# not true;;
- : bool = false
```

   Conditional expressions are constructed in the usual way:

```
# let x = "abcd" in
  let y = "abd" in
```

```
  String.length(if x < y then x else y);;
- : int = 4
```

4. The type of 8-bit characters, `char`, consisting of values such as `'a'`, `'E'`, `'\n'`. The `char_of_int : int -> char` and `int_of_char : char -> int` convert between the two using the ASCII mapping.[1]

5. The type of strings of 8-characters, `string`. String constants are enclosed within double-quotation marks, as in `"Hello, World!"`. The following functions are defined:
   a) Indexing the individual characters of a string, like `s.[i]`, which returns the `i`-th character of the string `s`.
   b) The infix operator for string concatenation, `(^) : string -> string -> string`.
   c) The function `String.length : string -> int` which returns the length of a string.

6. The type `unit` of the single value `()`.

```
# ();;
- : unit = ()
```

The following derived data types are also very useful:

1. The type of lists of elements with a common type `'a`: `'a list = [] | :: of 'a * 'a list` . They may be compactly represented using the list notation, as in `[ 1; 5; 8; 9 ]`. The following pre-defined functions are relevant:
   a) The functions returning the head and tail of a list:
      i. `List.hd : 'a list -> 'a`, and
      ii. `List.tl : 'a list -> 'a list`.

      Both functions throw an exception when applied to the empty list.
   b) The function returning the length of a list, `List.length : 'a list -> int`.
   c) The infix operator to concatenate two lists, `(@) : 'a list -> 'a list -> 'a list`.
   d) `List.map : ('a -> 'b)-> 'a list -> 'b list`.
   e) `List.fold_left : ('a -> 'b -> 'a)-> 'a -> 'b list -> 'a`.
   f) `List.fold_right : ('a -> 'b -> 'b)-> 'a list -> 'b -> 'b`.
   g) `List.filter : ('a -> bool)-> 'a list -> 'a list`.
   h) `List.find : ('a -> bool)-> 'a list -> 'a`.
   i) `List.exists : ('a -> bool)-> 'a list -> bool`.
   j) `List.for_all : ('a -> bool)-> 'a list -> bool`.

2. The type of n-way products, `'a1 * 'a2 * ··· * 'an`. These can be constructed as `(e1, e2, ..., en)`. The `unit` type can be regarded as a degenerate 0-way product.

[1] Citation needed.

a) The functions returning the first and second elements of a pair, `fst : 'a * 'b -> 'a` and `snd : 'a * 'b -> 'b`, respectively.

5 Syntactic Trivia

1. Comments:
   a) `(* This is a comment. *)`
   b) `(* Comments can be nested. (*Like this. *)*)`
2. Two forms of `let` bindings exist:
   a) Let declarations:
      ```
      # let v = e;;
      # ...
      ```
      Here, the variable `v` is bound to the result of evaluating `e` in the rest of the program. It works in both the REPL and in freestanding files. Redeclarations result in shadowing, without warning.
   b) Let expressions:
      ```
      # let v = e1 in e2;;
      ```
      Here, the variable `v` is bound to the result of evaluating `e1` while evaluating `e2`. Notably, `v` is not bound while evaluating the first sub-expression `e1`. Once again, nested redeclarations of `v` result in shadowing, without warnings.

   For example,
   ```
   # let v = 3;;
   val v : int = 3
   # let v = v + v;;
   val v : int = 6
   # v;;
   - : int = 6
   # let v = 3 in (let v = v + v in v + 2) + v
   - : int = 11
   ```
3. Functions can be:
   a) Non-recursive, as in `let f x = e`. The previous value of `f`, if any, is used while evaluating `e`. For example,
      ```
      # let f x = x + 3;;
      val f : int -> int = <fun>
      # let f x = (f x) + 3;;
      val f : int -> int = <fun>
      # f 3;;
      - : int = 9
      ```
   b) Recursive, as in `let rec f x = e`. References to the variable `f` within `e` result in recursive calls.

c) Mutually recursive, as in:

```
# let rec even x = match x with 0 -> true | _ -> odd (x - 1)
    and
          odd x = match x with 0 -> false | _ -> even (x -
              1);;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
# even 0;;
- : bool = true
# even 5;;
- : bool = false
# odd 5;;
- : bool = true
```

d) Or of the anonymous non-recursive kind:

```
# let f = fun x -> x @ [ 4; 5 ];;
val f : int list -> int list = <fun>
```

4. Custom data types, as in:

```
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree;
```

Note that type names begin with a lower-case letter, while constructors begin with an upper-case letter. Constructors are not functions. See Xavier Leroy's justification.

5. One can match against syntactic patterns, similar to the following piece of code:

```
# let l = [ 1; 2; 3 ];;
val l : int list = [1; 2; 3]
# match l with
  | [] -> 0
  | a :: b :: [] -> 5
  | a :: b :: _ -> 7
  | _ -> 2;;
- : int = 7
```

Partial matches are fine, as in the following, but the compiler will complain:

```
# let f x = match x with true -> false;;
Line 1, characters 10-36:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
false
val f : bool -> bool = <fun>
# f true;;
- : bool = false
```

```
# f false;;
Exception: "Match_failure //toplevel//:1:10"
Called from file "toplevel/toploop.ml", line 212, characters 17-27
```

The keyword `function` is similar to `fun`, but with built-in pattern-matching:

```
# let f = function [] -> 0 | hd :: tl -> hd;;
val f : int list -> int = <fun>
# f (3 :: []);;
- : int = 3
# let g = function [] -> 0;;
Line 1, characters 8-24:
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
_::_
val g : 'a list -> int = <fun>
```

6 Diagnosing Failures with Debuggers