OCaml Style Guide

Fall 2017

Contents

1 Introduction 1

2 Naming 1

3 Formatting 2

  3.1 Indenting ......................................................... 2
  3.2 Line Breaks ......................................................... 4
  3.3 Parentheses ......................................................... 4
  3.4 Spacing ............................................................. 6

4 Pattern Matching 6

5 Verbosity 10

6 Acknowledgments 11

1 Introduction

While the OCaml language supports functional, imperative, and object-oriented programming constructs, this style guide only covers aspects of OCaml related to the functional programming features used in CS 17.

All the OCaml code you write in CS 17 must follow all the guidelines spelled out in this document. Furthermore, all the OCaml code you write beyond CS 17 should continue to follow the guidelines spelled out in this document, and/or those spelled out by OCaml’s developers at INRIA: http://caml.inria.fr/resources/doc/guides/guidelines.en.html.

2 Naming

The following are the identifier naming guidelines that are followed by the OCaml library. You should abide by these conventions in CS 17:
<table>
<thead>
<tr>
<th>Identifier Type</th>
<th>Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants and Procedures</td>
<td>Initial letter must be lower case.</td>
</tr>
<tr>
<td></td>
<td>Use underscores for multiword names.</td>
</tr>
<tr>
<td></td>
<td>Example: <code>km_to_miles</code></td>
</tr>
<tr>
<td>Predicates</td>
<td>You cannot end predicate names with a ?.</td>
</tr>
<tr>
<td></td>
<td>Instead, use <code>P</code>, like <code>memberP</code>.</td>
</tr>
<tr>
<td></td>
<td>Or <code>is</code>, like <code>is_empty</code>.</td>
</tr>
<tr>
<td>Types</td>
<td>All letters should be lower case.</td>
</tr>
<tr>
<td></td>
<td>Use underscores for multiword names.</td>
</tr>
<tr>
<td></td>
<td>Example: <code>priority_queue</code></td>
</tr>
<tr>
<td>Constructors</td>
<td>Initial letter must be upper case.</td>
</tr>
<tr>
<td></td>
<td>Use embedded caps for multiword names.</td>
</tr>
<tr>
<td></td>
<td>Historic exceptions are true and false.</td>
</tr>
<tr>
<td></td>
<td>Examples: <code>Node</code>, <code>EmptyQueue</code></td>
</tr>
<tr>
<td>Module Types</td>
<td>Initial letter must be upper case.</td>
</tr>
<tr>
<td></td>
<td>Use all caps, and underscores for multiword names.</td>
</tr>
<tr>
<td></td>
<td>Example: <code>PRIORITY_QUEUE</code></td>
</tr>
<tr>
<td>Modules</td>
<td>Initial letter must be upper case.</td>
</tr>
<tr>
<td></td>
<td>Use embedded caps for multiword names.</td>
</tr>
<tr>
<td></td>
<td>Example: <code>PriorityQueue</code></td>
</tr>
<tr>
<td>Functors</td>
<td>Same as module type convention.</td>
</tr>
<tr>
<td></td>
<td>Example: <code>PriorityQueue</code></td>
</tr>
</tbody>
</table>

3  Formatting

In most sections of an OCaml program, how you use white space is not mandated by the compiler. Its use, then, comes down to style.

3.1  Indenting

If you happen to use `ocaml-top`, it indents your code for you, so you can gloss over this section. But if you use another text editor, you must follow the conventions set forth in this section.

- **Indent most lines by two spaces.** Lines that indent code should do so by two spaces more than the previous line of code. For example:

  ```ocaml
  (* Bad *)
  let proc (foo : int) (bar : int) : int =
  foo * bar
  
  (* Good *)
  let proc (foo : int) (bar : int) : int =
  foo * bar
  ```

  Lines that wrap around should indent by more. For example:
let x = ("This is a really long string that is intentionally" ^
"longer than 80 characters, and hence cannot possibly" ^
"fit on one, or even two, lines.")

- **How to indent nested let expressions.** Nested let expressions should not be indented. Nor should let and expressions.

  (* Bad *)
  let x = exp1 in
  let y = exp2 in
  x + y

  (* Good *)
  let x = exp1 in
  let y = exp2 in
  x + y

  (* Also Good *)
  let x = exp1
  and y = exp2 in
  x + y

  (* Still Good *)
  let x = very_very_long_exp
  in x + y

  In this last example, it makes sense to put in at the start of a line because it could get lost in the shuffle at the end of a very very long expression.

- **How to indent match expressions.** Align match expressions as follows:

  match expr with
  | pat1 -> ...
  | pat2 -> ...

- **How to indent if expressions.** Indent if expressions using one of these options, depending on the length of the expressions:

  if exp1 then exp2 else exp3

  if exp1 then exp2
  else exp3

  if exp1 then
  exp2
  else exp3

  In the first example, the expressions are short enough to all fit on one line. In the second example, the first two expressions are short enough to fit on the same line. In the third example, the expressions are too long to even fit two of them on the same line.
Here is an example of nested \texttt{if} expressions:

\begin{verbatim}
if \texttt{exp1} then \texttt{exp2}
else if \texttt{exp3} then \texttt{exp4}
else \texttt{exp5}
\end{verbatim}

- \textbf{How to indent comments}. Comments should be indented to the level of the line of code to which the comment refers—usually, the line that follows the comment.

3.2 Line Breaks

If you have a long line of code with parallel structure, breaking it up can improve readability. Pattern matching is a perfect example of this. The following is functional code, but it’s hard to read:

\begin{verbatim}
match \texttt{aloi} with
| \texttt{[]} -> -15 | \texttt{hd :: tl} when \texttt{hd} > 0 -> 17 * \texttt{hd} | \texttt{hd :: tl} -> 0
\end{verbatim}

This is much better:

\begin{verbatim}
match \texttt{aloi} with
| \texttt{[]} -> -15
| \texttt{hd :: tl} when \texttt{hd} > 0 -> 17 * \texttt{hd}
| \texttt{hd :: tl} -> 0
\end{verbatim}

This same idea also arises when you are dealing with a compound data structure such as a tree. Understanding its form when it is written linearly instead of structurally can be difficult:

\begin{verbatim}
let \texttt{my_tree} = \texttt{Node (17, Node (18, Node (19, \texttt{Leaf}, \texttt{Leaf}),}
\texttt{ Node (22, Node (33, \texttt{Leaf}, \texttt{Leaf}), \texttt{Leaf}}), \texttt{Leaf})
\end{verbatim}

So breaking it up across multiple lines that reflect its structure is usually a good idea:

\begin{verbatim}
let \texttt{my_tree} = \texttt{Node (17,}
\texttt{ Node (18,}
\texttt{ Node (19, \texttt{Leaf}, \texttt{Leaf}),}
\texttt{ Node (22,}
\texttt{ Node (33, \texttt{Leaf}, \texttt{Leaf}),}
\texttt{ \texttt{Leaf}}),
\texttt{ \texttt{Leaf}})
\end{verbatim}

3.3 Parentheses

- \textbf{Use parentheses sparingly}. Like in math, and unlike in Racket, the following expressions are equivalent in OCaml:

\begin{verbatim}
4
\end{verbatim}
In these expressions, the use of parentheses is redundant. They do not change the semantics, and hence should be used sparingly (if at all).

But, like in Racket, parentheses in OCaml often do have semantic content. They are used to construct tuples, to override built-in operator precedence, and to group structures into functor arguments. In these cases, parentheses are necessary, and hence, must be used.

Here is an example of using parentheses to override built-in operator precedence:

(* Bad *)
x+y * z+a

(* Good *)
(x + y) * (z + a)

Spaces (and indentation) do not achieve the effect of parentheses. The former of these two expressions computes \( x + (y * z) + a \), which does not appear to be what was intended.

- **Parenthesize to help indentation.** Automated indentation algorithms are often assisted by parentheses. Consider the following:

```ocaml
let x = "Long line..." ^
       "Another long line."

let x = ("Long line..." ^
       "Another long line.")
```

The latter informs an editor that the long line spills over onto another long line, so that the editor can indent it properly.

- **Wrap nested match expressions with parentheses.** It is almost always necessary to parenthesize nested match expressions, so it is good practice to do so by default. For example, the following code snippet looks like it should evaluate to 3 when \( \text{digit1} \) is bound to 7, but it doesn’t! (Instead, it evaluates to 3 when \( \text{digit2} \) is bound to 7.)

```ocaml
(* Bad *)
match digit1 with
| 1 -> match digit2 with
  | 5 -> 1
  | 6 -> 2
| 7 -> 3

Proper use of parentheses disambiguates this code:
(* Good *)

match digit1 with
| 1 -> (match digit2 with
               | 5 -> 1
               | 6 -> 2)
| 7 -> 3

- **OCaml is not Java.** Parentheses should never appear on a line by themselves, nor should they be the first graphical character on a line.

### 3.4 Spacing

*The space bar is your friend!* Don’t be afraid to press it. It is a nice big key, so it is easy to find. Use it.

1. Surround infix operators by spaces. Write this `hd :: tl`, not this `hd::tl`, and `x + y`, not `x+y`.

2. Insert spaces before and after the `:` in type annotation: e.g., `(17 : int)`. (Why? Because `:` is an infix operator!)

3. Insert a space after a pipe in a type definition or a `match` expression.

```ocaml
type season =
 | Fall
 | Winter
 | Spring
 | Summer
```

4. Insert a space after a type constructor: e.g., `Hearts 7`.

5. Insert a space before `;;` at the end of each definition.

On the other hand, just like you should not use too many parentheses, you should not insert too many spaces! For example, do not surround a procedure’s arguments by parentheses and/or spaces.

(* Bad *)

my_procedure ( arg_1 ) ( arg_2 )

(* Good *)

my_procedure arg_1 arg_2

### 4 Pattern Matching

- **Pattern matching should always follow the structure of the data.** Suppose you define a variant type, such as:
type train_car =
| Engine
| Boxcar of int (* capacity *)
| Caboose

When you pattern match on data of type train_car, your match expression should follow the structure of the data in the same order, like this:

match train with
| Engine -> ...
| Boxcar n -> ...
| Caboose -> ...

Also, your pattern matching should be exhaustive. This is not complete:

match train with
| Engine -> ...
| Caboose -> ...

If your pattern matching is incomplete, the OCaml compiler will issue a warning, as follows: “Warning: this pattern-matching is not exhaustive.” Treat such warnings as bugs!

In some cases the compiler will flag a match as incomplete when actually it isn’t. This is because the compiler is not smart enough to infer that all possible cases have been covered. For example,

match t with
| Leaf -> Node (Leaf, datum, Leaf)
| Node (left, x, right) when (datum < x) -> Node (insert datum left, x, right)
| Node (left, x, right) when (datum > x) -> Node (left, x, insert datum right)
| Node (left, x, right) when (datum = x) -> Node (left, x, right)

The incorrect way to eliminate this warning would be to simply add a catch-all case, as follows:

| Leaf -> Node (Leaf, datum, Leaf)
| Node (left, x, right) when (datum < x) -> Node (insert datum left, x, right)
| Node (left, x, right) when (datum > x) -> Node (left, x, insert datum right)
| Node (left, x, right) when (datum = x) -> Node (left, x, right)
| _ -> failwith "EMF" (* Bad *)

Here, EMF stands for “Exhaustive Match Failure.

The correct way to eliminate an compiler warning that arises from a failure to pattern match exhaustively is to add an explicit unguarded match as follows:

| Leaf -> Node (Leaf, datum, Leaf)
| Node (left, x, right) when (datum < x) -> Node (insert datum left, x, right)
| Node (left, x, right) when (datum > x) -> Node (left, x, insert datum right)
| Node (left, x, right) when (datum = x) -> Node (left, x, right)
| Node (left, x, right) -> failwith "EMF" (* Good *)
This latter approach is preferable because it preserves the ability of the compiler to flag unmatched cases, which is one of the key features of OCaml.

In summary, never appease the compiler by inserting a “catch-all”. Doing so negates the power of the compiler, and will impede your power to debug and extend your code.

- **Use pattern matching for selection.** Instead of using `fst` and `snd` to deconstruct a tuple, use pattern matching. For example:

  ```ocaml
type posn = float * float

    (* Bad *)
    let p = some_posn in
    let x = fst p in
    let y = snd p in
    x +. y

    (* Good *)
    let x, y = some_posn in
    x +. y

Similarly, records should be deconstructed using pattern matching:

  ```ocaml
type circle = {center : posn, radius : float}

    (* Bad *)
    let circ = some_circle in
    let c = circ.center in
    let r = circ.radius in
    let x = fst c in
    let y = snd c in
    r *. (x +. y)

    (* Good *)
    let {center = (x, y); radius = r} = some_circle in
    r *. (x +. y)

You should also steer away from using `List.hd` and `List.tl` in favor of pattern matching.

- **Pattern match using as few `match` expressions as possible.** Rather than nest match expressions, you can often pattern match against a tuple.

  ```ocaml
  (* Bad *)
  match month with
  | Jan -> (match day with
          | 1 -> "Happy New Year"
          | _ -> ")
  | Mar -> (match day with
          | 14 -> "Happy Pi Day"
          | _ -> "")

```
| Oct -> (match day with
| 10 -> "Happy Metric Day"
| _ -> "")

(* Good *)
match month, day with
| Jan, 1 -> "Happy New Year"
| Mar, 14 -> "Happy Pi Day"
| Oct, 10 -> "Happy Metric Day"
| _ -> ""

• Never use only one pipe in a match expression. There is never a need for only one pipe in a match expression. In such cases, prefer let.

(* Bad *)
match card with
| Hearts n -> ...

(* Good *)
let Hearts n = card in
...

• Pattern match a procedure’s formal arguments when possible. In OCaml, compound types are deconstructed via pattern matching, using match expressions, let expressions, or by deconstructing a procedure’s formal arguments. You should use the latter option if you need only the constituents of a procedure’s formal arguments, and have no use for the arguments as a whole. In this example, the procedure’s formal arguments are tuples:

(* Bad *)
let f arg1 arg2 =
  let x = fst arg1 in
  let y = snd arg1 in
  let z = fst arg2 in
  ...

(* Good *)
let f (x, y) (z, _) =
  ...

There is no need to name arg1 and arg2 here, since all that is needed are their constituents. Likewise, in this example, where the procedure’s formal arguments are records:

(* Bad *)
let f arg1 arg2 =
  let x = arg1.field1 in
  let y = arg1.field2 in
  let z = arg2.field3 in
  ...

(* Good *)
let f (arg1, arg2) =
  ...

...
(* Good *)
let f {field1 = x; field2 = y} {field3 = z; field4 = _} = ...

5 Verbosity

- **Simplify if expressions.** There are a number of equivalent ways to express the same conditional logic. In almost all cases, shorter expressions are preferred:

<table>
<thead>
<tr>
<th>Verbose</th>
<th>Concise</th>
</tr>
</thead>
<tbody>
<tr>
<td>if expr then true else false expr</td>
<td>expr</td>
</tr>
<tr>
<td>if expr then false else true not expr</td>
<td></td>
</tr>
<tr>
<td>if expr then expr else false expr</td>
<td>expr</td>
</tr>
<tr>
<td>if not expr then x else y if expr then y else x</td>
<td></td>
</tr>
<tr>
<td>if x then true else y x</td>
<td></td>
</tr>
<tr>
<td>if x then y else false x &amp;&amp; y</td>
<td></td>
</tr>
<tr>
<td>if x then false else y not x &amp;&amp; y</td>
<td></td>
</tr>
<tr>
<td>if x then y else true not x</td>
<td></td>
</tr>
</tbody>
</table>

When an *if* expression is used for argument selection, it can be embedded within a procedure application to improve readability, as follows:

(* Duplication of f a b applications *)
if c then f a b x
else f a b y

(* Can be eliminated by embedding the if *)
f a b (if c then x else y)

- **Don’t rewrap procedures.** When applying a procedure to another procedure, don’t rewrap the procedure if it already does what you need it to do. Here are two examples:

(* Verbose *)
List.map (fun x -> sqrt x) [1.0; 4.0; 9.0; 16.0]

(* Concise *)
List.map sqrt [1.0; 4.0; 9.0; 16.0]

(* Verbose *)
List.fold_left (fun x y -> x + y) 0

(* Concise *)
List.fold_left ( + ) 0

- **Don’t misuse match expressions.** Do not use *match* when you mean *if!*
(*) Bad *)
  match expr with
  | true -> x
  | false -> y

(*) Good *)
  if expr then x else y

(*) Bad *)
  match expr with
  | c -> x (* c is a constant *)
  | _ -> y

(*) Good *)
  if e = c then x else y

- Don’t overuse match expressions. Do not bind expressions unnecessarily. For example, do not bind something here, simply pattern match on it:

  (*) Bad *)
  let x = something in
  match x with
  ...

  (*) Good *)
  match something with
  ...

Similarly, do not use match to pattern match when let is enough:

  (*) Bad *)
  let x = match expr with (y, z) -> y

  (*) Good *)
  let (x, _) = expr

- Don’t reinvent the wheel. Built in to the OCaml library are a great number of ready-made procedures and data structures. You should use them, unless of course an assignment expressly forbids it!

  For example, when writing a procedure that recursively walks down a list, think fold! Some other data structures have similar folding procedures; use them when they are available.

6 Acknowledgments

Much of this style guide was copied from parts of the CIS120 style guide at the University of Pennsylvania, which in turn copied much of its content of the CS312 style guide at Cornell University.
Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17 document by filling out the anonymous feedback form:

http://cs.brown.edu/courses/cs017/feedback