CS51: OCAML STYLE GUIDE

STUART M. SHIEBER

Contents

Date: January 29, 2018.

This style guide is reworked from a long line of style guides for courses at Princeton, University of Pennsylvania, and Cornell, including Cornell CS 312, U Penn CIS 500 and CIS 120, and Princeton COS 326. All this shows the great power of recursion. (Also, the joke about recursion was stolen from COS 326.)

This guide provides some simple rules of good programming style, both general and OCaml-specific, developed for the Harvard course CS51. The rules presented here tend to follow from a small set of underlying principles.

Consistency: Similar decisions should be made within similar contexts.

Brevity: "Everything should be made as simple as possible, but no simpler." (attr. Albert Einstein)

Clarity: Code should be chosen so as to communicate clearly to the human reader.

Transparency: Appearance should summarize and reflect structure.

Like all rules, those below are not to be followed slavishly but seen as instances of these underlying principles. These principles may sometimes be in conflict, in which case judgment is required in finding the best way to write the code. This is one of the many ways in which programming is an art, not (just) a science.

This guide is not complete. For more recommendations, from the OCaml developers themselves, see the official OCaml guidelines.

1. Formatting

Formatting concerns the layout of the text of a program on the screen or page, such issues as vertical alignments and indentation, line breaks and whitespace. To allow for repeatable formatting, code is typically presented with a fixed-width font in which all characters including spaces take up the same horizontal pitch.

1.1. **No tab characters.** You may feel inclined to use tab characters (ASCII 0x09) to align text. Do not do so; use spaces instead. The width of a tab is not uniform across all renderings, and what looks good on your machine may look terrible on another's, especially if you have mixed spaces and tabs. Some text editors map the tab key to a sequence of spaces rather than a tab character; in this case, it's fine to use the tab key.

1.2. **80 column limit.** No line of code should extend beyond 80 characters long. Using more than 80 columns typically causes your code to wrap around to the next line, which is devastating to readability.

1.3. **No needless blank lines.** The obvious way to stay within the 80 character limit imposed by the rule above is to press the enter key every once in a while. However, blank lines should only be used at major logical breaks in a program, for instance, between value declarations, especially between function declarations. Often it is not necessary to have blank lines between other declarations unless you are separating the different types of declarations (such as structures, types, exceptions and values). Unless function declarations within a `let` block are long, there should be no blank lines within a `let` block. There should absolutely never be a blank line within an expression.

1.4. **Use parentheses sparely.** Parentheses have many purposes in OCaml, including constructing tuples, grouping sequences of side-effect expressions, forcing higher precedence on an expression for parsing, and grouping structures for functor arguments. Clearly, parentheses must be used with care, as they force the reader to disambiguate the intended purpose of the parentheses, making code more difficult to understand. You should therefore only use parentheses when necessary or when doing so improves readability.

✗
```
let x = function1 (arg1) (arg2) (function2 (arg3)) (arg4)
```

✓
```
let x = function1 arg1 arg2 (function2 arg3) arg4
```

On the other hand, it is often useful to add parentheses to help indentation algorithms, as in this example:

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

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

Similarly, wrapping `match` expressions in parentheses helps avoid a common (and confusing) error that you get when you have a nested `match` expression.

Parentheses should never appear on a line by themselves, nor should they be the first visible character; parentheses do not serve the same purpose as brackets do in C or Java.

1.5. **Spacing for operators and delimiters.** Operators (arithmetic operators like + and *, the typing operator `:`, type forming operators like * and ->, etc.) should be surrounded by spaces. Delimiters like the list item delimiter `;` and the tuple element delimiter `,` are followed but not preceded by a space.

✓
```
let f (x : int) : int * int = 3 * x - 1, 3 * x + 1 ;;
```

✗
```
let f (x: int): int*int = 3* x-1, 3* x+1 ;;
```

Judgement can be applied to vary from these rules for clarity's sake, for instance, when emphasizing precedence.

✓
```
let f (x : int) : int * int = 3*x - 1, 3*x + 1 ;;
```

1.6. **Indentation.** Indentation should be used to encode the block structure of the code as described in the following sections. It is typical to indent by two xor four spaces. Choose one system for indentation, and be consistent throughout your code.

1.6.1. *Indenting `if` expressions.* Indent `if` expressions using one of the following methods, depending on the sizes of the expressions. For very short *then* and *else* branches, a single line may be sufficient.

✓
```
if exp1 then veryshortexp2 else veryshortexp3
```

When the branches are too long for a single line, move the else onto its own line.

✓
```
if exp1 then exp2
else exp3
```

This style lends itself nicely to nested conditionals.

✓
```
if exp1 then shortexp2
else if exp3 then shortexp4
else if exp5 then shortexp6
else exp8
```

For very long *then* or *else* branches, the branch expression can be indented and use multiple lines.

✓
```
if exp1 then
  longexp2
else shortexp3
```

✓
```
if exp1 then
  longexp2
else
  longexp3
```

Some use an alternative conditional layout, with the `then` and `else` keywords starting their own lines.

✗
```
if exp1
then exp2
else exp3
```

This approach is less attractive for nested conditionals and long branches.

1.6.2. *Indenting `let` expressions.* Indent the body of a `let` expression the same as the `let` keyword itself.

✓
```
let x = definition in
code_that_uses_x
```

This is an exception to the rule of further indenting subexpression blocks to manifest the nesting structure.

✗
```
let x = definition in
  code_that_uses_x
```

The intention is that `let` definitions be thought of like mathematical assumptions that are listed before their use, leading to the following attractive indentation for multiple definitions:

```
let x = x_definition in
let y = y_definition in
let z = z_definition in
block_that_uses_all_the_defined_notions
```

1.6.3. *Indenting `match` expressions.* Indent `match` expressions so that the patterns are aligned with the `match` keyword, always including the initial (optional) `|`, as follows:

```
match expr with
| first_pattern -> ...
| second_pattern -> ...
```

Some disfavor aligning the arrows in a match, arguing that it makes the code harder to maintain. However, where there is strong parallelism among the patterns, this alignment (and others) can make the parallelism easier to see, and hence the code easier to understand. Use your judgement.

2. Documentation

2.1. **Comments before code.** Comments go above the code they reference. Consider the following:

✗
```
let sum = List.fold_left (+) 0
(* Sums a list of integers. *)
```

✓
```
(* Sums a list of integers. *)
let sum = List.fold_left (+) 0
```

The latter is the better style, although you may find some source code that uses the first. Comments should be indented to the level of the line of code that follows the comment.

2.2. **Comment length should match abstraction level.** Long comments, usually focused on overall structure and function for a program, tend to appear at the top of a file. In that type of comment, you should explain the overall design of the code and reference any sources that have more information about the algorithms or data structures. Comments can document the design and structure of a class at some length. For individual functions or methods comments should state the invariants, the non-obvious, or any references that have more information about the code. Avoid comments that merely restate the code they reference or state the obvious. All other comments in the file should be as short as possible; after all, brevity is the soul of wit. Rarely should you need to comment within a function; expressive variable naming should be enough.

2.3. **Multi-line commenting.** There are several styles for demarcating multi-line comments in OCaml. Some use this style:

```
(* This is one of those rare but long comments
 * that need to span multiple lines because
 * the code is unusually complex and requires
```

Token	*Convention*	*Example*
Variables and functions	Symbolic or initial lower case. Use underscores for multiword names.	`get_item`
Constructors	Initial upper case. Use embedded caps for multiword names. Historical exceptions are `true` and `false`.	`Node`, `EmptyQueue`
Types	All lower case. Use underscores for multiword names.	`priority_queue`
Module Types	Initial upper case. Use embedded caps for multiword names.	`PriorityQueue`
Modules	Same as module type convention.	`PriorityQueue`
Functors	Same as module type convention.	`PriorityQueue`

TABLE 1. Naming conventions

```
 * extra explanation. *)
let complicated_function () = ...
```

arguing that the aligned asterisks demarcate the comment well when it is viewed without syntax highlighting. Others find this style heavy-handed and hard to maintain without good code editor support (for instance, emacs Tuareg mode doesn't support it well), leading to this alternative:

```
(* This is one of those rare but long comments
   that need to span multiple lines because
   the code is unusually complex and requires
   extra explanation.
 *)
let complicated_function () = ...
```

Whichever you use, be consistent.

3. Naming and declarations

3.1. **Naming conventions.** Table 1 provides the naming convention rules that are followed by OCaml libraries. You should follow them too. Some of these naming conventions are enforced by the compiler. For example, it is not possible to have the name of a variable start with an uppercase letter.

3.2. **Use meaningful names.** Variable names should describe what the variables are for, in the form of a word or sequence of words. By convention (Table 1) the words in a variable name are separated by underscores (`multi_word_name`), not (ironically) distinguished by camel case (`multiWordName`).

In short `let` blocks, one letter variable names can sometimes be appropriate. Often it is the case that a function used in a `fold`, `filter`, or `map` is bound to the name `f`. Here is an example with appropriate variable names:

```
let date = Unix.localtime (Unix.time ()) in
let minutes = date.Unix.tm_min in
```

```
let seconds = date.Unix.tm_min in
let f n = (n mod 3) = 0 in
List.filter f [minutes; seconds]
```

3.3. **Function declarations and type annotations.** Top-level functions and values should be declared with explicit type annotations to allow the compiler to verify the programmer's intentions. Use spaces around `:`, as with all operators.

✗
```
let incr x = x + 1
```

✓
```
let incr (x : int) : int = x + 1
```

When a function being declared has multiple arguments with complicated types, so that the declaration doesn't fit nicely on one line,

✗
```
let rec zip3 (x : 'a list) (y : 'b list) (z : 'c list) : ('a * 'b * 'c) list option =
  ...
```

one of the following indentation conventions can be used:

✓
```
let rec zip3 (x : 'a list)
             (y : 'b list)
             (z : 'c list)
           : ('a * 'b * 'c) list option =
  ...
```

✓
```
let rec zip3
          (x : 'a list)
          (y : 'b list)
          (z : 'c list)
        : ('a * 'b * 'c) list option =
  ...
```

3.4. **Avoid global mutable variables.** Mutable values, on the rare occasion that they are necessary at all, should be local to functions and almost never declared as a structure's value. Making a mutable value global causes many problems. First, an algorithm that mutates the value cannot be ensured that the value is consistent with the algorithm, as it might be modified outside the function or by a previous execution of the algorithm. Second, and more importantly, having global mutable values makes it more likely that your code is nonreentrant. Without proper knowledge of the ramifications, declaring global mutable values can easily lead not only to bad design but also to incorrect code.

3.5. **When to rename variables.** You should rarely need to rename values: in fact, this is a sure way to obfuscate code. Renaming a value should be backed up with a very good reason. One instance where renaming a variable is both common and reasonable is aliasing structures. In these cases, other structures used by functions within the current structure are aliased to one or two letter variables at the top of the `struct` block. This serves two purposes: it shortens the name of the structure and it documents the structures you use. Here is an example:

```
module H = Hashtbl
module L = List
module A = Array
...
```

3.6. **Order of declarations in a structure.** When declaring elements in a file (or nested module) you first alias the structures you intend to use, then declare the types, then define exceptions, and finally list all the value declarations for the structure. Separating each of these sections with a blank line is good practice unless the whole is quite short. Here is an example:

```
module L = List
type foo = int
exception InternalError
let first list = L.nth list 0
```

Every declaration within the structure should be indented the same amount.

4. Pattern matching

4.1. **No incomplete pattern matches.** Incomplete pattern matches are flagged with compiler warnings, and you should avoid them. In fact, it's best if your code generates no warnings at all.

4.2. **Pattern match in the function arguments when possible.** Tuples, records, and algebraic datatypes can be deconstructed using pattern matching. If you simply deconstruct the function argument before you do anything useful, it is better to pattern match in the function argument itself. Consider these examples:

✗
```
let f arg1 arg2 =
  let x = fst arg1 in
  let y = snd arg1 in
  let z = fst arg2 in
  ...
```

✓
```
let f (x, y) (z, _) =
  ...
```

✗
```
let f arg1 =
  let x = arg1.foo in
  let y = arg1.bar in
  let baz = arg1.baz in
  ...
```

✓
```
let f {foo = x; bar = y; baz} =
  ...
```

See also the discussion of extraneous match expressions in let definitions in Section 4.4.

The exception to pattern matching in a function argument is when the patterns are atomic values. You should only deconstruct structured values in function arguments. If you want to

pattern match against a specific atomic value, use an explicit `match` expression or an `if` expression. Consider the following:

✗
```
let rec fact = function
  | 0 -> 1
  | n -> n * fact (n - 1)
```

✓
```
let rec fact n =
  if n = 0 then 1
  else n * fact (n - 1)
```

We include this rule because there are too many errors that can occur when you don't do this exactly right.

4.3. **Pattern match with as few `match` expressions as necessary.** Rather than nesting `match` expressions, you can sometimes combine them by pattern matching against a tuple. Of course, this doesn't work if one of the nested `match` expressions matches against a value obtained from a branch in another `match` expression. Nevertheless, if all the values are independent of each other you should combine the values in a tuple and match against that. Here is an example:

✗
```
let d = Date.fromTimeLocal (Unix.time ()) in
match Date.month d with
| Date.Jan -> (match Date.day d with
                | 1 -> print "Happy New Year"
                | _ -> ())
| Date.Jul -> (match Date.day d with
                | 4 -> print "Happy Independence Day"
                | _ -> ())
| Date.Oct -> (match Date.day d with
                | 10 -> print "Happy Metric Day"
                | _ -> ())
```

✓
```
let d = Date.fromTimeLocal (Unix.time ()) in
match (Date.month d, Date.day d) with
| (Date.Jan, 1)  -> print "Happy New Year"
| (Date.Jul, 4)  -> print "Happy Independence Day"
| (Date.Oct, 10) -> print "Happy Metric Day"
| _ -> ()
```

(This example also provides a case where aligning arrows improves clarity.)

4.4. **Misusing `match` expressions.** The `match` expression is misused in two common situations. First, `match` should never be used with single atomic values in place of an `if` expression. (That's why `if` exists.). For instance,

✗

```
match e with
| true -> x
| false -> y
```

✓
```
if e then x else y
```

and

✗
```
match e with
| c -> x (* c is a constant value *)
| _ -> y
```

✓
```
if e = c then x else y
```

Second, `match` expressions should not be used when pattern matching within an enclosing expression (like `let`, `fun`, `function`) allows pattern-matching itself:

✗
```
let x = match expr with
        | y, z -> y
```

✓
```
let x, _ = expr
```

4.5. **Avoid using too many projection functions.** Frequently projecting a value from a record or tuple causes your code to become unreadable. This is especially a problem with tuple projection because the value is not documented by a mnemonic name. To prevent projections, you should use pattern matching with a function argument or a value declaration. Of course, using projections is okay as long as use is infrequent and the meaning is clearly understood from the context.

✗
```
let v = some_function () in
let x = fst v in
let y = snd v in
x + y
```

✓
```
let x, y = some_function () in
x + y
```

4.5.1. *Don't use `List.hd` or `List.tl` at all.* The functions `hd` and `tl` are used to deconstruct list types; however, they raise exceptions on certain arguments. You should never use these functions. In the case that you find it absolutely necessary to use these (something that probably won't ever happen), you should explicitly handle any exceptions that can be raised by these functions.

5. Verbosity

5.1. **Reuse code where possible.** The OCaml standard library has a great number of functions and data structures. Unless told otherwise, use them! Often students will recode `List.filter`, `List.map`, and similar functions. A more subtle situation for recoding is all the fold functions. Functions that recursively walk down lists should make vigorous use of `List.fold_left` or `List.fold_right`. Other data structures often have a fold function; use them whenever they are available. (Sometimes we will ask you to implement some constructs yourself rather than relying on a library function. In such cases, we'll specify that using library functions is not allowed.)

5.2. **Do not abuse `if` expressions.** Remember that the type of the condition in an `if` expression is `bool`. In general, the type of an `if` expression can be any `'a`, but in the case that the type is `bool`, you should not be using `if` at all. Consider the following:

✗
```
if e then true else false
```

✓
```
e
```

✗
```
if e then false else true
```

✓
```
not e
```

✗
```
if e then e else false
```

✓
```
e
```

✗
```
if x then true else y
```

✓
```
x || y
```

✗
```
if x then y else false
```

✓
```
x && y
```

✗
```
if x then false else y
```

✓
```
not x && y
```

Also problematic is overly complex conditions such as extraneous negation.

✗
```
if not e then x else y
```

✓
```
if e then y else x
```

The exception here is if the expression `y` is very long and complex, in which case it may be more readable to have it placed at the end of the `if` expression.

5.3. **Don't rewrap functions.** Don't fall for the misconception that functions passed as arguments have to start with `fun` or `function`, which leads to the extraneous rewrapping of functions like this:

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

Instead, just pass the function directly.

✓
```
List.map sqrt [1.0; 4.0; 9.0; 16.0]
```

You can even do this when the function is an infix binary operator, though you'll need to place the operator in parentheses.

✗
```
List.fold_left (fun x y -> x + y) 0
```

✓
```
List.fold_left (+) 0
```

5.4. **Avoid computing values twice.** When computing values more than once, you may be wasting CPU time (a design consideration) and making your program less clear (a style consideration) and harder to maintain (a consideration of both design and style). The best way to avoid computing things twice is to create a `let` expression and bind the computed value to a variable name. This has the added benefit of letting you document the purpose of the value with a well-chosen variable name, which means less commenting. On the other hand, not every computed sub-value needs to be `let`-bound.

✗
```
f (calc_score (if cond then val1 else val2))
  (calc_score (if cond then val1 else val2))
```

✓
```
let score = calc_score (if cond then val1 else val2) in
f score score
```

6. Other common infelicities

Here is a compilation of some other common infelicities to watch out for:

Bad	*Good*
`l::[]`	`[l]`
`length + 0`	`length`
`length * 1`	`length`
`big_expression * big_expression`	`let x = big_expression in x*x`
`if x then f a b c1 else f a b c2`	`f a b (if x then c1 else c2)`
`String.compare x y = 0`	`x = y`
`String.compare x y < 0`	`x < y`
`String.compare y x < 0`	`x > y`