Style guide for Standard ML programmers

COMP 105 course staff
(with help from CS 312 staff at Cornell)

Spring 2019

Contents

1 Introduction

Programming-language people care about concise, precise, and elegant code. We take style seriously. This document provides guidelines about what constitutes proper style for programs written in Standard ML. Proper style gives you some freedom in expressing code the way you want to express it while making it easy for the course staff to read your work. We owe a significant debt to another style guide which was used at Cornell in courses long taught by Greg Morrisett and Andrew Myers.

2 Basic courtesy

Code must fit in 80 columns.

Code may not contain tab characters. The width of a tab character varies with the machine, and what looks good on your machine may look terrible on mine. For indentation, use spaces.

Code must compile. Any code you submit must compile *without errors or warnings*. No excuses. Never submit anything that you have changed, no matter how small the change, without checking that it still compiles.

3 Comments

Minimize comments. Eliminate these sorts of comments:

- A comment that contains the same information as the code it refers to
- A comment that provides a narrative description of a sequence of events in a computation, like an algorithm
- A comment that provides information which could instead be conveyed by the name of a variable
- A comment that provides information which could instead be conveyed by the *type* of a variable or function
- A long comment in the middle of a definition

Write these sorts of comments instead:

- A comment that states a function's contract
- A comment that explains what a let-bound or lambda-bound name represents
- A comment which shows algebraic laws that a function obeys

If you wish to explain a design or refer to an external description of an algorithm or data structure, a long comment at the top of a block of code may be OK.

Multi-line Commenting. When code is printed on paper or rendered as PDF, it can be difficult or expensive to use color to distinguish code from comments. We therefore ask that if you write a large comment spanning multiple lines, you mark it in a way that is clearly distinct from code. We recommend on of two styles:

- Precede each line of the comment with a `*` as follows:

```
(* 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. *)
```

  This style can be easy to read but hard to edit without special support.

- A reasonable alternative is

```
(***********************************************
   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.
  ***********************************************)
```

Better still: don't write code that demands long comments.

4 Parentheses

Redundant parentheses are distracting, and they may mark you as an inexperienced ML programmer. But it's easy to be confused about when you need parentheses. Here's a checklist to tell you when you need parentheses around an expression or a pattern:

1. Is it an argument to a (possibly Curried) function, and if so, is it more than a single token?
2. Is it an infix expression that should be parenthesized because it is the argument of *another* infix operator?[1]
3. Are you forming a tuple?
4. Are you parenthesizing an expression involving `fn`, `case`, or `handle`?

If the answer to any of these questions is yes, use parentheses. Otherwise, you almost certainly don't need them—so get rid of them!

Here are some common places where parentheses are always redundant:

- Around a name
- Around a literal constant

[1] It is often a good idea to parenthesize nested infix expressions even when, strictly speaking, the parentheses are redundant. Beyond the usual operations for $+$, $-$, $\times$, and `div`, nobody can remember which operators have precedence over which other operators. If you write nested infix applications using more exotic operators, consider parentheses.

- Around the element of a tuple
- Around the condition in an `if`-expression

```
   (* ALL THESE EXAMPLES ARE WRONG *)

VAR (x)          ((theta tau1), (theta tau2))

LIT (0)          if (null xs) then ... else ...

                 fun f ((x::xs), (y::ys)) = ...
```

We will mark down submissions that include redundant parentheses.

5 Indentation and layout

Use blank lines carefully. Blank lines are useful primarily at top level, to separate distinct definitions or groups of related definitions.

Blank lines can improve readability, but there is a tradeoff: Your editor may display only 40–50 lines at once. If you need to read and understand a large function all at once, you mustn't overuse blank lines.

Unless function definitions within a `let` block are long, there should be no blank lines within a `let` block. There should never be a blank line within an expression or between two clauses that define the same function.

Indent by two or three spaces. Be consistent. You can try indenting by four spaces, but you may run out of horizontal space and have to break up expressions that should be written on a single line.

Avoid breaking expressions over multiple lines Here's an example of what not to do:

```
(* HORRID *)
   fun mid   (x, y, z) = y
   fun right (x, y, z) = z

   fun euclid (m, n) : int * int * int =
     if n = 0 then (b 1, b 0, m)
     else (mid (euclid (n, m mod n)), u - (m div n) *
             (euclid (n, m mod n)), right (euclid (n, m mod n)))
```

If you have to split a record or a tuple over multiple lines, use "MacQueen notation:" the opening and closing brackets, together with the separating commas, appear leftmost:

```
(* STILL BAD, BUT THE TUPLE LOOKS GOOD *)
   fun mid   (x, y, z) = y
   fun right (x, y, z) = z

   fun euclid (m, n) : int * int * int =
     if n = 0 then (b 1, b 0, m)
     else ( mid   (euclid (n, m mod n))
```

```
          , left  (euclid (n, m mod n)) - (m div n) * mid (euclid (n, m mod n))
          , right (euclid (n, m mod n))
          )
       (* the layout of the tuple is OK, but not much else is right *)
```

A better plan, especially for this example, is to use a `let` construct to name intermediate results:

```
(* GOOD *)
   fun euclid (m, n) : int * int * int =
     if n = 0 then
       (b 1, b 0, m)
     else
       let val (q, r)    = (m div n, m mod n)  (* quotient/remainder *)
           val (u, v, g) = euclid (n, r)
       in  (v, u - q * v, g)
       end
```

Pay attention to horizontal alignment When you do have to break up a long expression, align the parts carefully. Don't break lines at operators or applications that are deeply nested in the abstract syntax. Here are several acceptable examples:

```
val x = "long string ... " ^
        "another long string"

val x = "long string ... " ^
  "another long string"

val x = "long string ... " ^
          "another long string"
```

Often the best way to break up an expression that won't fit on one line is to name and `val`-bind a subexpression.

Use horizontal space. When you are writing function applications and infix operations, use horizontal space liberally.

- Use space between a function and its argument
- Use space between an infix operator and its arguments
- Use space between elements of a tuple

Examples:

```
1 + max (height l, height r)  (* right *)

1+max(height l,height r)      (* four times WRONG *)
```

5.1 Indentation of particular syntactic forms

5.1.1 Definitions of types and values

Type, fun, and val bindings should be indented so that = signs line up when reasonable:

```
type exp  = Ast.exp
type loc  = Ast.name_or_mem
type name = string
type hint = string
type convention = string
type procname   = string
type label      = string
```

And another example:

```
fun eprint r s = E.errorRegionPrt (srcmap, r) s
fun errorf r   = Printf.kprintf (fn s => eprint r s)
val pointerty  = Types.Bits (Metrics.pointersize metrics)
val wordty     = Types.Bits (Metrics.wordsize metrics)
val vfp        = Vfp.mk (Metrics.wordsize metrics)
val memsize    = Metrics.memsize metrics
```

5.1.2 Blocks of definitions, including function definitions

Blocks of definitions such as are found in `let`... `in`... `end`, `struct`... `end`, or `sig`... `end` should be indented as follows:

```
fun foo bar =
  let val p = 4
      val q = 8
  in  bar * (p + q)
  end
```

In rare cases it may be acceptable to introduce additional vertical space:

```
fun foo bar =
  let (* this form should be rare *)
    val p = 4
    val q = 8
  in
    bar * (p + q)
  end
```

5.1.3 Case expressions

Indent *case expressions* with a line break *before* `of` and so the `f` in `of` aligns with vertical bars. When reasonable, the arrows should line up as well:

```
case expr
  of pattern1 => exp1
   | pat2     => exp2
```

Please don't put `of` on the same line as `case`.

5.1.4 If expressions

If expressions can be indented in many acceptable ways.

- Short expressions should appear on one line:

```
if exp1 then exp2 else exp3   (* preferred *)
```

- Very long expressions should appear on four lines:

```
if exp1 then       (* good for very long expressions *)
  exp2
else
  exp3
```

- If the condition is also very long, place `then` on a line by itself, with the same indentation as the `if` it goes with:

```
if exp1 which is very long      (* very long conditions *)
then
  exp2
else
  exp3
```

- Here are some less preferred but still acceptable alternatives:

```
(* acceptable for medium-size expressions *)
if exp1 then exp2
else exp3

(* also acceptable for medium-size expressions *)
if exp1 then exp2
        else exp3

(* also acceptable for medium-size expressions *)
if exp1
then exp2
else exp3
```

Nested ifs should use nested indentation, except when the nesting forms a *sequence* of if-then-else checks. At that point it's a judgment call. Here are some acceptable non-nested formats for sequences of if-then-else:

```
if exp1      then exp2
else if exp3 then exp4
else if exp5 then exp6
```

```
else exp8

     if exp1 then exp2
else if exp3 then exp4
else if exp5 then exp6
else exp8

if exp1 then
  exp2
else if exp3 then
  exp4
else if exp5 then
  exp6
else
  exp8
```

6 Naming and Declarations

Use standard naming conventions. By using the standard naming conventions for Standard ML, you provide instant information about what a name stands for. Please use these conventions, which have been established by the SML standard basis and by the Standard ML of New Jersey libraries:

Token	*SML naming convention*	*Example*
Variable	A variable is symbolic or begins with a small letter. Multiword names use embedded capital letters. *A function is just another variable*, so the same conventions apply to functions.	`getItem`
Constructor	A constructor is symbolic or uses all capital letters. Multiword names use underscores. (Milner's contructors `nil`, `true`, and `false` are grandfathered.) Symbolic constructors are rare.	`NODE` `EMPTY_QUEUE`
Type	A type is written with small letters. Multiword names use underscores.	`priority_queue`
Signature	A signature is written with all capital letters. Multiword names use underscores.	`PRIORITY_QUEUE`
Structure	A structure begins with a capital letter. Multiword names use embedded capital letters.	`PriorityQueue`
Functor	A functor uses the same naming convention as a structure. Its name may begin with `Mk` or end with `Fn`.	`PriorityQueueFn`

Sadly, these conventions are not enforced by the compiler.

Use meaningful names. The name of a variable or function should say something about what the variable represents or what the function computes. In small scopes, short variable names are acceptable, as in C; here is an example:

```
let
  val d = Date.fromTimeLocal(Time.now())
  val m = Date.minute d
  val s = Date.second d
  fun f n = (n mod 3) = 0
in
  List.filter f [m,s]
end
```

Conventional names include `f` for a function, `s` for a string, `n` for an integer, `x` or `y` for a value of unknown type, `a` or `b` for a value of unknown type, `xs` or `ys` for a list, `xss` for a list of lists, and so on.

Definitional interpreters have their own conventions: `x` for a name, `e` for an expression, `v` for a value, `d` for a definition, `tau` for a type, `rho` for an environment, and many others.

Name a predicate so you know what truth means. If you write a function that returns a Boolean, make sure that the name of the function tells you the meaning of a `true` or `false` result:

```
fun checkList xs = ...                (* BAD *)

fun lengthIsEven xs = ...             (* GOOD *)
fun listIsEmpty xs  = ...             (* GOOD *)
```

The "good" examples don't need any documentation beyond their names and types.

Name a general function after the thing it returns. For example, if you want to convert a `bar` to a `foo`, use "of," not "to."

```
val fooOfBar : bar -> foo             (* GOOD *)
val fooOf    : bar -> foo             (* GOOD *)

val barToFoo : bar -> foo             (* BAD *)
val bar2foo  : bar -> foo             (* BAD *)
```

Remember the cost of naming intermediate results. When intermediate results are short and simple, it's usually clearer not to name them:

```
(* BAD *)
let val x = TextIO.inputLine TextIO.stdIn
in  case x of ...
end

(* GOOD *)
case TextIO.inputLine TextIO.stdIn
  of ...
```

```
(* BAD (provided y is not a large expression) *)
let val x = y * y in x + z end

(* GOOD *)
y * y + z
```

7 Shorter is sweeter

Don't duplicate the initial basis. Especially don't write your own versions of `filter`, `map`, and so on. When you can, use `foldr`.

Simplify conditionals that return Booleans. If you have a background in C or C++, it's easy to forget that if-then-else is an *expression* and can be simplified. And if a conditional returns a Boolean, chances are it *should* be simplified:

Bad	*Good*
`if e then true else false`	`e`
`if e then false else true`	`not e`
`if beta then beta else false`	`beta`
`if x then true else y`	`x orelse y`
`if x then y else false`	`x andalso y`
`if x then false else y`	`not x andalso y`
`if x then y else true`	`not x orelse y`

Simplify everything. More short sweetness.

Bad	*Good*
`l :: nil`	`[l]`
`l :: []`	`[l]`
`length + 0`	`length`
`length * 1`	`length`
`big * big`	`let val x = big in x * x end`
`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) = EQUAL`	`x = y`
`String.compare (x, y) = LESS`	`x < y`
`String.compare (x, y) = GREATER`	`x > y`
`Int.compare (x, y) = EQUAL`	`x = y`
`Int.compare (x, y) = LESS`	`x < y`
`Int.compare (x, y) = GREATER`	`x > y`
`Int.sign x = 1`	`x < 0`
`Int.sign x = 0`	`x = 0`
`Int.sign x = 1`	`x > 0`

Don't rewrap functions in lambdas. When passing a function as an argument to another function, don't rewrap the function in a fresh lambda (spelled "`fn`").[2] Here's an example:

```
List.map (fn x => Math.sqrt x) [1.0, 4.0, 9.0, 16.0]    (* WRONG *)

List.map Math.sqrt [1.0, 4.0, 9.0, 16.0]                (* RIGHT *)
```

Beginners often rewrap infix binary operators, especially when using a fold. To avoid rewrapping a binary operator, use the `op` keyword, as in the following example:

```
foldl (fn (x,y) => x + y) 0    (* BAD *)

foldl (op +) 0                 (* GOOD *)
```

Avoid nesting let expressions. You can and should put multiple `val` and `fun` bindings in a single `let` expression. Nested `let` should almost always be replaced with a single `let`.

```
let val x = 42                        (* BAD *)
in
  let val y = x + 101
  in  x + y
  end
end

let val x = 42                        (* GOOD *)
    val y = x + 101
in  x + y
end
```

8 Pattern matching and case analysis

Never write an incomplete pattern match. Our special version of Moscow ML rejects code with incomplete pattern matches. Other compilers will give warnings. In this class, any file that includes an incomplete pattern match earns No Credit. If you are deliberately setting out to write code that doesn't cover every possible constructor, because you know that not all constructors are used, match against the unused constructors anyway, and on the right-hand side, `raise Match` explicitly.

Prefer pattern matching to equality. Using `=` restricts your polymorphic functions to operate only on values of "equality type." This restriction is a slight embarrassment to the designers of ML. Pattern matching carries no such restriction, so when possible, use pattern matching instead of `=`:

```
if l = [] then ... else ...             (* NOT ACCEPTABLE *)


case l                                  (* ACCEPTABLE *)
  of []     => ...
```

[2] Creating an anonymous function to wrap another function is called *eta-expansion*, and it is occasionally required to work around the dreaded *value restriction*.

```
    | _ :: _ => ...

if null l then ... else ...              (* SPLENDID *)
```

The "splendid" version uses the standard library function `null`.

Write your own projection functions. It is almost never correct for a beginner to use `#1`, `#2`, `#foo`, `#bar`, and so on. Use pattern matching instead. And auxiliary functions are OK:

```
fun fst (x, _) = x                       (* GOOD *)
fun snd (_, y) = y
```

Commenting such functions would be execrable.

Use pattern matching early. If you simply deconstruct a function's argument before you do anything useful, it is better to pattern match in the function argument. Consider these examples:

```
fun f arg1 arg2 =                        (* BAD *)
  let val (x, y) = arg1
      val (z, _) = arg2
  in  ...
  end

fun f (x, y) (z, _) =                    (* GOOD *)
  ...
```

In the bad code, the names `arg1` and `arg2` are a blight on the landscape. They are meaningless and exist only to be matched against one time. But you could do even worse:

```
fun f arg1 arg2 =                        (* OUTRIGHT WRONG *)
  let val x = #1 arg1 (* may not typecheck *)
      val y = #2 arg1 (* may not typecheck *)
      val z = #1 arg2 (* may not typecheck *)
  in  ...
  end
```

Prefer `fun` to `case`.

```
fun length []      = 0                   (* GOOD *)
  | length (x::xs) = 1 + length xs


fun length l =                           (* BAD *)
  case l
    of []      => 0
     | x :: xs => 1 + length xs
```

Instead of nesting `case`, match a tuple. Instead of nesting matches and `case` expressions, you can sometimes combine them by pattern matching against a tuple. Here is an example:

```
(* UGLY *)
     let val d = Date.fromTimeLocal(Time.now())
     in  case Date.month d
           of Date.Jan => (case Date.day d
                             of 1  => SOME "New Year"
                              | _  => NONE)
            | Date.Jul => (case Date.day d
                             of 4  => SOME "Independence Day"
                              | _  => NONE)
            | Date.Oct => (case Date.day d
                             of 10 => SOME "Metric Day"
                              | _  => NONE)
     end

(* NOT UGLY *)
     let val d = Date.fromTimeLocal(Time.now())
     in  case (Date.month d, Date.day d)
           of (Date.Jan,  1) => SOME "New Year"
            | (Date.Jul,  4) => SOME "Independence Day"
            | (Date.Oct, 10) => SOME "Metric Day"
            | _ => NONE
     end
```

Avoid `valOf`, `hd`, and `tl`. Use pattern matching instead of `valOf`, `hd`, and `tl`. These functions raise exceptions on certain inputs, and it is usually easy to achieve the same effect with pattern matching.

Don't use `case` for a job `val` can handle. When a `case` expression has a single alternative, that's a sign that you probably should use `val` instead:

```
val x = case expr of (y,z) => y    (* BAD *)

val (x, _) = expr                  (* GOOD *)
```

9 Common mistakes to avoid

The material in this section comes from the course supplement to Ullman.

Avoid `open`. Ullman sometimes abbreviates by opening structures, e.g., `open TextIO`. **Never do this**—it is bad enough to open structures in the standard basis, but if you open other structures, your code will be hopelessly difficulty to maintain. The `open` directive is the devil's tool.

Instead of using `open`, abbreviate structure names as needed. For example, after

```
structure T = TextIO
```

you can use `T.openIn`, and so on, without (much) danger of confusion.

Avoid #1 and #2 Some beginners pick up the idea that a good way to get the second element of a pair `p` is to write `#2 p`. This style is not idiomatic or readable, and it can confuse the type checker. The proper way to handle pairs is by pattern matching, so

```
fun first  (x, _) = x
fun second (_, y) = y
```

is preferred, and not

```
fun bogus_first  p = #1 p   (* WRONG *)
fun bogus_second p = #2 p
```

(For reasons I don't want to discuss, these versions don't even type-check.)

If your pair or tuple is not an argument to a function, use `val` to do the pattern matching:

```
val (x, y) = lookup_pair mumble
```

But usually you can include matching in ordinary fun matching.

Avoid polymorphic equality. It is almost never correct to ask if an expression is equal to a constructor; use pattern matching instead. Or even better, if the type is predefined, you may be able to use a function from the initial basis.

```
xs = nil    (* WRONG *)

(case xs of [] => true | _ :: _ => false)  (* BETTER *)

null xs     (* BEST *)



e = NONE (* WRONG *)

(case e of NONE => true | SOME _ => false) (* BETTER *)

not (isSome e)  (* BEST *)
```

Avoid the semicolon. When you're using an interpreter interactively, you need a semicolon to terminate a top-level expression or definition. But *the semicolon should almost never appear in your code.* Don't use a semicolon unless you are deliberately sequencing imperative code. Ullman's book is full of unnecessary semicolons. Don't emulate it.

Avoid redundant parentheses around conditions. Redundant parentheses are such a problem that this style guide has a whole section on them. But one of the most common mistakes is to forget that you should *never* parenthesize a condition that appears between `if` and `then`.

10 Helpful code you might easily overlook

10.1 Essential library modules

Learn some of the essential library modules. The library modules `String`, `Char`, and `Int` contain a high proportion of helpful functions. Module `TextIO` has a lot more junk in it, but if you wind up writing anything serious in ML, you'll need it.

10.2 ML's answer to the null pointer: option types

Let's suppose you want to represent a value, except the value might not actually be known. For example, maybe a homework grade is represented as a string, except if a grade hasn't been submitted. Or maybe a square on a chessboard contains a piece, except the square might be empty. This problem comes up so often that the initial basis for ML has a special type constructor called `option`, which handles all situations of this kind, polymorphically.

The definition of `option` is

```
datatype 'a option = NONE | SOME of 'a
```

and it is already defined when you start the interactive system. *You need not and should not define it yourself.*

Examples:

```
- datatype chesspiece = K | Q | R | N | B | P
- type square = chesspiece option
- val empty : square = NONE
- val lower_left : square = SOME R
- fun play piece = SOME piece : square;
> val play = fn : chesspiece -> chesspiece option

- SOME true;
> val it = SOME true : bool option
- SOME 37;
> val it = SOME 37 : int option

- SOME "fish" = SOME "fowl";
> val it = false : bool
- SOME "fish" = NONE;
> val it = false : bool
- "fish" = NONE;
! Toplevel input:
! "fish" = NONE;
!          ^^^^
! Type clash: expression of type
!   'a option
! cannot be made to have type
!   string
```

The `option` type is covered in Ullman on pages 111-113, 208, etc.

There is an `Option` structure in the initial basis, and it contains a couple of handy functions like `Option.map` and `Option.join`.

10.3 Immutable arrays, also called vectors

Although Ullman describes the mutable `Array` structure in Chapter 7, he doesn't cover the immutable `Vector` structure except for a couple of pages deep in Chapter 9. Like an array, a vector offers constant-time access to an array of elements, but a vector is not mutable. Because of its immutability, `Vector` is often preferred. It is especially flexible when initialized with `Vector.tabulate`. Here's the signature:

```
signature VECTOR =
  sig
    eqtype 'a vector
    val maxLen : int
    val fromList : 'a list -> 'a vector
    val tabulate : int * (int -> 'a) -> 'a vector
    val length : 'a vector -> int
    val sub : 'a vector * int -> 'a
    val extract : 'a vector * int * int option -> 'a vector
    val concat : 'a vector list -> 'a vector
    val app : ('a -> unit) -> 'a vector -> unit
    val foldl : ('a * 'b -> 'b) -> 'b -> 'a vector -> 'b
    val foldr : ('a * 'b -> 'b) -> 'b -> 'a vector -> 'b
    val appi : (int * 'a -> unit) -> 'a vector * int * int option -> unit
    val foldli : (int * 'a * 'b -> 'b)
                 -> 'b -> 'a vector * int * int option -> 'b
    val foldri : (int * 'a * 'b -> 'b)
                 -> 'b -> 'a vector * int * int option -> 'b
  end
```

11 Awkward ML problems

Understand Curried and tupled. In ML, any function you define can be in Curried form (like `map` or `foldr` or `List.filter`) or it can take a tuple as a single argument (like `List.take` or `List.drop`). I find curried functions more readable, but the Standard Basis Library and the infix conventions push toward tupled functions. Pick a convention you like and stick with it.

Know how to write type annotations. In Standard ML, it can be extremely helpful to write the types of your functions and have them checked by the compiler, but you are hosed: There is no good syntax for doing so. The best you can hope for is something like the following:

```
fun foo x = x+1
val foo = foo : int -> int  (* explicit type*)
```

or the following

```
val foo : int -> int = fn x => x + 1
```

The latter alternative does not work well with Curried functions.

Be careful using shared mutable state. Shared mutable state is a tool to be used carefully. If you must have some, be prepared to justify it. Globally visible `ref` cells, in particular, are almost never justified (but some globally visible, mutable abstractions can be justified.)

Know your folds. Every kind of "container" data structure can have a fold function, as can every algebraic data type. These fold functions play the same role as "iterators" in object-oriented languages like C++ and Java. Know how to use your folds.