SML Style Guide

Last Revised: Saturday 22nd August, 2020

It is an old observation that the best writers sometimes disregard the rules of rhetoric. When they do so, however, the reader will usually find in the sentence some compensating merit, attained at the cost of the violation. Unless he is certain of doing as well, he will probably do best to follow the rules. (William Strunk, *The Elements of Style*)

Vigorous writing is concise. A sentence should contain no unnecessary words, a paragraph no unnecessary sentences, for the same reason that a drawing should have no unnecessary lines and a machine no unnecessary parts. This requires not that the writer make all sentences short or avoid all detail and treat subjects only in outline, but that every word tell.
(William Strunk, *The Elements of Style*)

Programs must be written for programmers to read, and only incidentally for machines to execute.
(Abelson and Sussman, *Structure and Interpretation of Computer Programs*)

Contents

1 Introduction

Programs are written for people to read. After transformation, programs can be executed by machines, but the concrete syntax of a high-level language is a tool for humans. In this sense, writing a program is the same as writing in English: it is an effort to communicate an idea. And like writing in English, there are many unequal ways to communicate the same idea—to write code that matches the same specification. A sense of style distinguishes the good programs from the bad. The grammar of programming languages is enforced automatically, but style is chosen by humans.

This document consists of a small number of rules that, when followed, make code easier to read, and therefore easier to understand, write, and fix.

Disclaimer If the rule is about something you have not seen yet in the course, come back to it later.

2 Rules

2.1 Write code to be read.

2.1.1 Comment your code.

Some lines of code are written in a specific way for a non-obvious reason. Such lines should be paired with a short comment, explaining what decisions were made in writing them. Not every line of code deserves comment—too many comments can be as confusing as too few.

2.1.2 Annotate types.

SML has a powerful type inference mechanism, but it's often hard to read code without explicit type annotations. Let your reader know what's going on. Explicitly asserting the type of expressions also constrains the typechecker, and may limit the scope of otherwise confusing error messages.

2.1.3 Do not write long lines.

People lose focus when confronted with long lines of code, reguardless of content: don't risk confusing your reader needlessly. As a rule, do not write lines of code with more than 80 characters. Many text editors are set to soft-wrap or truncate at 80 characters, so violating this limit will make your code render badly.

2.1.4 Emphasize structure with white space.

Use new lines, spaces, and indentation consistently to make the internal structure of the program externally obvious. Indent with space characters, not tab characters, so that your white space is always rendered as you intended.

An easy way to do this is to partition the keywords into opening keywords and closing keywords as follows:

> **Opening:** `sig`, `struct`, `let`, `in`
> **Closing:** `end`, `let`, `in`

Opening keywords are followed immediately by a new line. The text that appears below them is at one deeper level of indentation. Closing keywords are immediately preceded by a new line, and they appear at one more shallow level of indentation than the text that precedes them.

For example,

```
fun f g x = let val (_, y) = g x 7
in
  y end
```

is harder to read than

```
fun f g x =
    let
      val (_, y) = g x 7
    in
      y
    end
```

because the let-in-end block is not indented under the function, the code between `let` and `in` is not at the same depth of indentation, and the `val` keyword is on the same line as the `let` keyword.

2.1.5 Give variables relevant names.

Use concise variable names that reflect how the variable is used. Short variable names are good when appropriate—i.e. `v1` instead of `velocity_of_particle_1`—but can introduce ambiguity.

There are some conventions on variable names. Variables that stand for any function are often named `f`, `g`, or `h`. Variables that stand for any list are often named `l`, or `L`. These are often subscripted to `L1`, `L2`, etc.

2.1.6 Do not shadow variables.

Shadowing variables cleverly can often shorten code, but does so at the cost of making it extremely hard to read. Someone reading your code should never wonder which instance of a variable with the same name they're currently considering.

2.1.7 Use datatypes.

Datatypes are a powerful tool to make your code more readable with very little cost. Do not use strings or integers (e.g. returning -1 for an error case) to represent a certain return value. Use datatypes instead.

2.1.8 Follow capitalization conventions.

Values and variables should be all lower case, possibly using underscores to separate words. Exceptions and structures should have the first letter in upper case and the rest in camel case.

Signatures names should be all upper case. Structure names should reflect the signature that they ascribe to, as well as hint at the abstraction function that they use to implement their signature.

```
signature STACK =
sig
  type 'a stack
  exception EmptyStack
  val empty : 'a stack
  val pop : 'a stack -> 'a * 'a stack
  val push : 'a stack -> 'a -> 'a stack
end

structure ListStack : STACK =
struct
  type 'a stack = 'a list
  exception EmptyStack

```

```
15    val empty = []
16
17    fun pop [] = raise EmptyStack
18      | pop (x :: l) = (x, l)
19
20    fun push s x = x :: s
21  end
```

2.1.9 Use parentheses carefully.

The meaning of under-parenthesized expressions is often not clear for human readers even if it has a unique parsing for the interpreter. Add parentheses to emphasize associations, but remove those that only add visual clutter. Adding parentheses does not necessarily add clarity.

2.2 Match the code to the idea.

Ideas have a structure to them, and code that implements with an idea should have a matching structure.

2.2.1 Write helper functions.

Write helper functions for small tasks that you do more than once. Use helper functions to isolate irrelevant details from pieces of code and to avoid repeating code.

Before writing a helper function, write a very careful specification for it. If the specification ends up being the same as the specification for the function calling the helper, you probably don't need it. If you change the type to add arguments, make sure that you actually use them.

2.2.2 Use pattern matching.

Recursive datatypes capture inductively defined structures; pattern matching lets you inspect them precisely. Elegantly defined case statements can express a lot of logic clearly in a small space. Try to flatten case statements instead of nesting them.

For example, consider writing a function that tests if the first two elements of a list of `int`s are 1. A first pass might be

```
1  fun ones l =
2    case List.nth(l,0)
3     of 1 =>
4        (case List.nth(l,1)
5          of 1 => true
6           | _ => false)
7      | _ => false
```

This performs the desired test, but is written in very poor style. The nested case statements can be safely flattened by testing the pair of the first elements at once, giving

```
1  fun ones l =
2      case (List.nth(l,0), List.nth(l,1))
3       of (1,1) => true
4        | _ => false
```

This case statement is exactly equivalent to using the boolean operator `andalso`, though, so this could be written even more clearly as

```
1  fun ones l = (List.nth(l,0) = 1) andalso (List.nth(l,1) = 1)
```

While concise, this implementation unnecessarily forces a boolean view of `l` instead of working on the structure of lists. If we instead respect the type of `l`, we pattern match and write the function

```
fun ones l =
  case l
    of 1 :: 1 :: _ => true
     | _ => false
```

2.2.3 Use case-statements to direct the flow of control.

Case statements, particularly on tuples, let you very elegant describe exactly what you want your program to do. The following code fragment arguably could be written with nested if-statements, but would be much less clear.

```
fun sublist_bp (i : int, j : int, L : int list) : int list =
  let
    val len = length L
    val err =
        case(len, i > j, i < 0 orelse j < 0, i >= len orelse j >= len)
         of (0,_,_,_) => SOME "cannot take a sublist of an empty list"
          | (_,true,_,_) => SOME "start index greater than end index"
          | (_,_,true,_) => SOME "both indices must be non-negative"
          | (_,_,_,true) => SOME "end index greater than list length"
          | _ => NONE
  in
    case err
     of NONE => sublist (i,j,L)
      | SOME err => raise Fail err
  end
```

2.2.4 Generalize problems when needed.

It's often the case that more general problems are easier to solve as specific instances. Don't be afraid to make helper functions with additional arguments that you can use.

2.2.5 Hide irrelevant internal details.

Use let-statements to hide helper functions or generalizations you only use in one function—other parts of the program don't need to have access to them.

2.3 Be concise.

Write concise code, but don't sacrifice clarity or elegance for concision. Don't be afraid to write clever code, but always write clear clever code.

2.3.1 If statements.

If the return type of an if-statement is a boolean, it can probably be reduced to an expression. Note that `if e then true else false` is the same as `e`, and `if e then false else true` is the same as `not e`.

2.3.2 Use library functions.

Make the most of the libraries available to you, and don't reimplement functions that they offer.

2.3.3 Use higher order functions.

You can often reuse code to great effect by parameterizing it on a function. Write functions as higher order when you can take advantage of it. You can often use specific instances of higher order functions to quickly implement seemingly complicated things.

2.4 Respect the type of expressions.

The type system is an incredibly powerful tool for describing what programs do: let the types guide how you program. Don't transform values of one type to another unless you need to.

Don't force types that are more expressive to act like less expressive types. For example, it happens to be the case that there are two list constructors and two values of type `bool`, but lists are far more expressive than booleans.

2.5 Revise your code.

Like all writing, programming is a process of revision. The first code that you write down that typechecks and passes simple tests is often completely unreadable. Keep all the old revisions around until you're happy so that you're not afraid to throw the current version out and start over.

2.6 Balance performance and style.

Performance can be a stylistic concern on its own: a convoluted solution that takes exponential time is worse than a direct one that takes linear time.

On the other hand, sometimes the most elegant way to write a function is drastically slower than a slightly less elegant form. It sometimes may be more important to be elegant than fast, but there is a balance there as well. Be cautious of small syntactic changes with profound semantic repercussions.

2.6.1 Name partial results to avoid repeating work.

If you need the result of a computation in more than one place, use a let-statement or a case-statement to name the result instead of computing it multiple times.

The fragment of code in section 2.2.3 binds the identifier `L` to the length of the argument list rather than computing it several times. Since computing the length of a list takes linear time, this saves a decent amount of work.