SML Style Guide

Parallel and Sequential Data Structures and Algorithms, 15-210 (Spring 2014)

January 13th, 2014

1 Preface

We will grade the style of your code, so keep your style consistent, readable and simple! Here are some suggestions to that end, although keep in mind that you certainly don't need to strictly adhere to all of these to produce well styled code, and you haven't necessarily produced well styled code just by following all the rules.

2 General Guidelines

- **No lines over 80 columns.** This is far more important than many people think it is. Follow this as a hard rule. This is even more important in this class because autolab will wrap lines that are longer than 80 characters and make it much more difficult for us to read your code. Since this is no longer an intro-level course, we will deduct points once we see lines over 80 characters long.
- **Never use tabs.** Change your editor settings to always replace tab characters with spaces. Use 2 spaces for each tab. Some prefer to indent 4 spaces after `fun` declarations, or other places. Whatever you choose, be consistent!
- **Comment when necessary; no less, no more.** Ideally, well-chosen (not necessarily long) variable names should make the purpose of individual lines obvious; too many comments breaks the flow of the code and makes it hard to read. *We can read SML*, so don't put comments that just describe exactly what each line of code does. Instead put comments that describe your algorithm or particular intricacies of your code at a higher level. The purpose of comments is to make it easier to understand the code, so comments that say the exact same thing as the code are not only useless, but also increase the volume of the code and make it harder to read!
- **Avoid large, multi-stage functions.** Even if these are well-commented, instead, use multiple short functions with concise explanations above the `fun` declaration.
- **Comments should stand out from code.** This means that multi-line comments have a leading asterisk on each line, like so:

```
(* This is a multi-line comment that doesn't exactly say
 * very much in terms of content, but illustrates pretty
 * well what we expect to see in your labs!
 *)

(* This is a short single-line comment above a function *)
fun add n k = ...
```

```
(* What is this I don't
(* even
*:(*)
```

- **Delete *dead code* from submissions.** Don't leave commented-out code in files you hand in, and don't leave in code that can never possibly be reached. Always take a moment to clean up your code before handing it in.
- **Follow the standard!** Don't invent any wacky, neo-modern styles in your 210 homework. TAs like plain and obvious code, so please be nice to us!

3 Spacing and Indentation

3.1 Case statements

You've been taught in 15-150 to use case statements by default. In general, this is a good habit. The styling of case statements should resemble one of the following:

```
case expr1
  of pattern1 => ...
   | pattern2 => ...

case expr1 of
    pattern1 => ...
  | pattern2 => ...
```

Spacing is **very important**. Always include a space before and after | and =>. Be mindful to avoid excessive indentation. In some cases (heh, heh) there may be other stylings that work better, but going with one of the plain, *uncreative* styles above is always a right choice.

3.2 Pattern-matching statements

When a function has few arguments and on the top level, there aren't too many cases to consider, instead of doing:

```
fun f data =
    case data
      of CTOR1 x => ...
       | CTOR2 x => ...
       | CTOR3 => ...
```

You can do:

```
fun f (CTOR1 x) = ...
  | f (CTOR2 x) = ...
  | f CTOR3 = ...
```

3.3 Nested statements

In SML, expressions can become very deeply nested and because of indentation, you can easily end up with very long lines. In particular, you may find yourself running into our 80-character line limit and then having badly formatted code going down the page but *smashed up* against the right side of the page (screen?).

Because of this, a good general strategy is to prefer adding newlines over making lines longer, and especially to prefer adding newlines *earlier* in your expressions. This way, you would not need to format the deeper-nested parts of your expressions.

For example, you may find yourself writing something like this:

```
let
  val enormousValueName = case niceDescriptiveName of
                            Foo => (case otherDescriptiveName of
                                      A blarg => let
                                                   val x = n+3
                                                 in
                                                   ...
                                                 end
                                    | B d => let
                                               val y = Seq.map
                                                         doStuff
                                                         blarg
                                             in
                                               ...
                                             end)
                          | Bar => let
                                     ...
                                   in
                                     ...
                                   end
in
  ...
end
```

Instead, this can be transformed into the much nicer:

```
let
  val enormousValueName =
    case niceDescriptiveName of
        Foo =>
          (case otherDescriptiveName of
               A blarg => let
                            val x = n+3
                          in
                            ...
                          end
             | B d => let
                        val y = Seq.map doStuff blarg
                      in
                        ...
                      end)
      | Bar => let
                 ...
               in
                 ...
               end
in
  ...
end
```

4 Conditionals

Yes, you can and *should* use if statements. This is a good idea when you are evaluating a strictly boolean expression. For example, instead of:

```
  case length s
   of 0 => ...
    | _ => ...

  case boolvar
   of true => ...
    | false => ...
```

It would look so much better if you did (respectively):

```
  if length s = 0
  then ...
  else ...

  if boolvar
  then ...
  else ...
```

A little note: it often helps readability if you put the smaller block of code in the `true` branch and the larger in the `false` branch (as above). Also, a boolean value can be negated with the SML unary `not` operator.

That said, **don't overuse conditionals!** Nesting `case` and `if` statements makes your code difficult to read and logically hard to follow. When you really need to, it's often better to express a multilevel conditional as a single-level case statement. For example, instead of:

```
if length s1 = 0
then ...
else if length s2 = 0
     then ...
     else let
            val (n1, n2) = (length s1, length s2)
          in
            ...
          end
```

Consider the much more concise alternative:

```
case (length s1, length s2) of
    (0, _) => ...
  | (_, 0) => ...
  | (n1, n2) => ...
```

5 Functions

5.1 Naming Functions

Use more `fun` declarations to give your functions a name binding for clarity and reuse. `fn` (anonymous functions) should only be used for short, self-explanatory functions. Instead of the oddly-shaped

```
val t = reduce (fn (x,y) =>
                   let
                     ...
                   in
                     ...
                   end)
               base
               (map (fn x =>
                        let
                          ...
                        in
                          ...
                        end)
                    s)
```

Move your functions out of the `val t` line:

```
fun combine (x, y) =
    ...
fun convert x =
    ...

val t = reduce combine base (map convert s)
```

5.2 Curried Functions

Most library functions are curried. This means you can easily make some useful helper functions by being *slightly clever*. For example, observe that some of the functions below are all partially applied.

```
fun f (S : 'a option seq seq) =
    let
      val get = nth S
      val copyScan = scan (fn (a, NONE) => a
                            | (_, b) => b) NONE
      val (s1', r1) = copyScan (get 2)
      ...
    in
      ...
    end
```

5.3 Making Code Concise

If you can make your code more concise, please do so. Avoid the following practices:

```
val a = Seq.reduce (fn (x,y) => x + y) 0 myInts
val b = Seq.map (fn x => someFunc x) mySeq
val c = if x > y then x else y
val d = case myOption of SOME(x) => x |  _ => raise Impossible
val e = if expr then false else true
```

Because the following is much more clear and concise!

```
val a = Seq.reduce (op+) 0 myInts (* Good trick to remember! *)
val b = Seq.map someFunc mySeq    (* No need for the anon *)
val c = Int.max (x,y)             (* Reinventing the wheel *)
val d = valOf myOption            (* Existing functions *)
val e = not expr                  (* Unary negation op *)
```

Furthermore, it is much easier to read a block of `val` declarations if they are separated by newlines like this:

```
val a = Seq.reduce (op+) 0 myInts

val b = Seq.map someFunc mySeq

val c = Int.max (x,y)

val d = valOf myOption

val e = not expr
```

Also, `a`, `b`, `c`, `d`, and `e` are not very good variable names. Neither are `s`, `s'`, `s''`, `s'''` and `s'"'`. One or two "primes" are OK; if you find yourself typing `s'"'`, it's time to use a better naming scheme. Better variable names could be:

```
val sum = Seq.reduce (op+) 0 myInts
val someFunced = Seq.map someFunc mySeq
val bigger = Int.max (x,y)
val foo = valOf myFooOption
val isntBar = not isBar
```

Or whatever makes sense in the context of the surrounding program.

Lastly, if you have comments between lines in your function, you should still add new lines to separate the comments from the previous line of code. So this:

```
(* ... *)
val x = ...
(* ... *)
val y = ...
(* ... *)
val z = ...
```

Should become:

```
(* ... *)
val x = ...

(* ... *)
val y = ...

(* ... *)
val z = ...
```

6 Further Help

That's all for now. Again, this is not an *exhaustive* style guide; just try to internalize as many of these practices as you can. As always, ask if you have questions, or come to office hours with your code if you'd like suggestions on how to clean it up.