SML Style Guide

Parallel and Sequential Data Structures and Algorithms, 15-210 (Fall 2015)

August 24th, 2015

1 General Guidelines

- **At most 80 characters per line.** This is far more important than many people realize. It is especially important in this class because Autolab likes to wrap lines over 80 characters, making them very difficult to read.
- **Never use tabs.** SML is not a 1-dimensional language. Many constructs (such as case statements and let-in-end blocks) require verticle alignment for readability. Since tabs might be rendered as 2, 4, or even 8 spaces, they are completely useless for precise verticle alignment. Many text editors have an option to automatically replace tabs with spaces – you should set your own editor to do the same.
- **Name your values well.** A little time spent picking good names can make a huge difference.
- **Strive for the "one-pass" rule.** Your code should be easily readable in a single pass from top to bottom. As a rule of thumb, this means that you should try to keep related pieces of code near one another. For example, helper functions should be written immediately above where they are used.
- **Stick to the standard.** Don't invent wacky, neo-modern conventions. The TAs like code which is plain and obvious, so be nice to them!
- **Be consistent.** Whatever you decide to do, stick to it. This is particularly important for indentation.

2 Comments

- Comment when necessary; no more, no less. Overcommenting is often worse than having no comments at all. Construct your comments carefully, and *tailor them to your audience*.
- Comments should be easily distinguishable from code. Multiline comments should have a vertically aligned leading asterisk on each line.

```
(* This is a nice multi-line comment which really
 * doesn't say anything but which is long just for
 * the sake of example. Notice how it's super
 * obvious that these lines are not code! *)
val foo = ...
val bar = ...
```

- To potentially make your grader's life a little easier, you might consider adding cost specifications in comments above functions:

```
(* Work: O(|s|)
 * Span: O(log|s|) *)
fun foo s = ...
```

3 Spacing

Use newlines to separate blocks of code that "go" together. You can also document each code block with a little description of what's going on. For example:

```
(* Determine overlap of every adjacent pair in s *)
fun pair i = (Seq.nth s i, Seq.nth s (i+1))
val adjPairs = Seq.tabulate pair (Seq.length s - 1)
fun overlap (x, y) = ...
val overlaps = Seq.map overlap adjPairs

(* Join the pair with the largest overlap *)
val idx = Seq.argmax Int.compare overlaps
val bestPair = Seq.nth adjPairs idx
val joined = join bestPair

(* Do more stuff ... *)
val foo = ...
val bar = ...
```

4 Functions

4.1 Helper Functions

- Try to design your helpers so that they may be used many times. A helper function which is used exactly once might not be necessary at all, unless it makes a significant impact on the readability of your code.
- Write your helper functions as near as possible to the location they are used. Don't be afraid of writing functions within functions, either.

4.2 Anonymous Functions

- Please **avoid writing multiline anonymous functions.** They often result in *nasty* indentation. For example:

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

We can write this instead as:

```
fun combine (x, y) =
  let ...
  in ...
  end
fun init x =
  let ...
  in ...
  end
val t = Seq.reduce combine base (Seq.map init s)
```

- Anonymous functions are often unnecessary if you pay attention to types carefully. For example,

```
Seq.reduce append
```

is much cleaner than

```
Seq.reduce (fn (x, y) => append (x, y))
```

4.3 Currying

Use it to your advantage! Here are some examples:

```
val sum = Seq.reduce op+ 0
fun repeat x = Seq.tabulate (fn _ => x)
val get = Seq.nth s
```

5 Parentheses

In SML, the canonical function application does not contain any parentheses.

```
val t = f(x) (* This is bad. *)
val t = f x (* This is good. *)
```

The latter uses fewer characters, is less cluttered, and is parsed *exactly the same.* This might be a hard habit to break, but trust us: fewer parentheses is a good thing.

6 Let-In-End Expressions

- Most let-in-end expressions should resemble one of the following. It is crucial that `let`, `in`, and `end` are vertically aligned.

```
let
  ...
in
  ...
end
```

```
let ...
in ...
end
```

- In some cases, a one-liner is convenient. This is acceptable as long as there is only one declaration inside the `let`.

```
fun delete (t, k) =
  let val (l, _, r) = split (t, k) in join (l, r) end
```

- You should avoid nesting as much as possible. For example, this code:

```
let
  val a = ...
  val b = ...
in
  let
    val c = ...
    val d = ...
  in
    e
  end
end
```

can be rewritten as:

```
let
  val a = ...
  val b = ...
  val c = ...
  val d = ...
in
  e
end
```

The exception to this rule is for functions inside of functions. The following is perfectly acceptable.

```
fun foo x =
  let
    fun bar y =
      let ...
      in ...
      end
  in
    ...
  end
```

7 Case Statements

- Most case statements should resemble one of the following. Once again, verticle alignment is crucial. The symbols | and => should always be surrounded by spaces on either side.

```
case ... of
  ... => ...
| ... => ...
```

```
case ...
 of ... => ...
  | ... => ...
```

- Single-liners are sometimes convenient:

```
val t = case ... of ... => ... | ... => ...
```

- Organize the branches of a case statement by their size, starting with the smallest branches. If you don't do this, then the small branches are easy to miss.

8 Conditionals

Yes, you *can* and *should* use conditional statements.

- Common designs for conditional statements are:

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

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

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

- Avoid excessive nesting, if you can. A scenario such as

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

  can be rewritten a little nicer as:

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

- As with case statements, put the smaller branch of the conditional first and the larger branch second. You can swap the branches simply by negating the boolean condition.
- **Never** write something resembling

```
if b then true else false
```

  This code is logically identical to just writing `b`. Similarly, the code

```
if b1 then true else b2
```

  is identical to `b1 orelse b2`.