ReasonML Style Guide
Fall 2019

Contents

1 Introduction 1
2 Naming 1

3 Formatting 1
  3.1 Indenting .................................................. 2
  3.2 Line Breaks .................................................. 3
  3.3 Parentheses & Braces ......................................... 4
  3.4 Spacing ...................................................... 4

4 Pattern Matching 5
5 Verbosity 8

6 Acknowledgments 10

1 Introduction

This style guide describes a style standard for CS 17’s subset of the language. All the ReasonML code you write in CS 17 must follow all the guidelines in this document.

2 Naming

The following are the identifier naming guidelines that are followed by the ReasonML library. You should abide by these conventions in CS 17:
### Identifier Type

<table>
<thead>
<tr>
<th>Identifier Type</th>
<th>Convention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constants and Procedures</td>
<td>Initial letter must be lower case. Use underscores for multiword names. Example: km_to_miles</td>
</tr>
<tr>
<td>Predicates</td>
<td>You cannot end predicate names with a ?. Instead, use P, like memberP. Or is, like is_empty.</td>
</tr>
<tr>
<td>Type Names</td>
<td>All letters should be lower case. Use underscores for multiword names. Example: priority_queue</td>
</tr>
<tr>
<td>Constructors</td>
<td>Initial letter must be upper case. Use embedded caps for multiword names. Historic exceptions are true and false. Examples: Node EmptyQueue</td>
</tr>
<tr>
<td>Module Types</td>
<td>Use all caps, and underscores for multiword names. Example: PRIORITY_QUEUE</td>
</tr>
<tr>
<td>Modules and File Names</td>
<td>Initial letter must be upper case. Use embedded caps for multiword names. Example: PriorityQueue</td>
</tr>
<tr>
<td>Functors</td>
<td>Same as module convention. Example: PriorityQueue</td>
</tr>
</tbody>
</table>

### 3 Formatting

In most sections of a ReasonML program, how you use white space is not mandated by the compiler. Its use, then, comes down to style.

#### 3.1 Indenting

If you enable the `editor.autoIndent` feature in your VS Code settings, it indents your code for you, so you can gloss over this section. But if you use another text editor, you must follow the conventions set forth in this section.

- **Indent most lines by two spaces.** Lines that indent code should do so by two spaces more than the previous line of code. For example:

```reasonml
/* Bad */
let proc = (foo: int, bar: int): int => {
  foo * bar
};

/* Good */
let proc = (foo: int, bar: int): int => {
  foo * bar
};
```

Lines that wrap around should indent by two spaces and remain aligned. For example:
let x = {
  "This is a really long string that is intentionally " ++
  "longer than 80 characters, and hence cannot possibly " ++
  "fit on one, or even two, lines."
};

- **How to indent** switch expressions. Align switch expressions as follows:

  ```reasonml```
  switch (expr) {
    | pat1 => ...
    | pat2 => ...
  }
  ```

  **Note:** The semicolon is required at the end of every declaration.

- **How to indent** if expressions. Indent if expressions using one of these options, depending on the length of the expressions:

  ```reasonml```
  if (exp1) {exp2} else {exp3};
  ```

  ```reasonml```
  if (exp1) {exp2}
  else {exp3};
  ```

  ```reasonml```
  if (exp1) {
    exp2
  } else {
    exp3
  }
  ```

  In the first example, the expressions are short enough to all fit on one line. In the second example, the first two expressions are short enough to fit on the same line. In the third example, the expressions are too long to even fit two of them on the same line.

  Here is an example of nested if expressions:

  ```reasonml```
  if (exp1) {exp2}
  else if (exp3) {exp4}
  else {exp5};
  ```

- **How to indent** comments. Comments should be indented to the level of the line of code to which the comment refers—usually, the line that follows the comment.

- **Auto-indentation.** If you’re using a different text editor, you can still take advantage of auto-indentation by going to [http://sketch.sh](http://sketch.sh) pasting your code in the box, and pressing Ctrl+Shift+I.
3.2 Line Breaks

If you have a long line of code with parallel structure, breaking it up can improve readability. Pattern matching is a perfect example of this. The following is functional code, but it’s hard to read:

```reasonml
switch (aloi) {
  | [] => -15
  | [hd, ...tl] when hd > 0 => 17 * hd
  | [hd, ...tl] => 0
};
```

This is much better:

```reasonml
switch (aloi) {
  | [] => -15
  | [hd, ...tl] when hd > 0 => 17 * hd
  | [hd, ...tl] => 0
};
```

This same idea also arises when you are dealing with a compound data structure such as a tree. Understanding its form when it is written linearly instead of structurally can be difficult:

```reasonml
let my_tree = { Node (17, Node (18, Node (19, Leaf, Leaf),
  Node (22, Node (33, Leaf, Leaf), Leaf)), Leaf)
};
```

So breaking it up across multiple lines that reflect its structure is usually a good idea:

```reasonml
let my_tree = { Node (17,
  Node (18,
    Node (19, Leaf, Leaf),
    Node (22,
      Node (33, Leaf, Leaf),
      Leaf)),
    Leaf)
};
```

3.3 Parentheses & Braces

- **Use parentheses sparingly.** As in math, and unlike in Racket, the following expressions are equivalent in ReasonML:

  ```reasonml
  17
  (17)
  ((17))
  (((17)))
  ((((17))))
  ```

In these expressions, the use of parentheses is redundant. They do not change the semantics, and hence should be used sparingly (if at all).
But, as in Racket, parentheses in ReasonML often do have semantic content. They are used to construct tuples, to override built-in operator precedence, to delimit function arguments, and to group structures into functor arguments. In these cases, parentheses are necessary, and hence, must be used.

Here is an example of using parentheses to override built-in operator precedence:

```reasonml
/* Bad */
x + y * z + a
/* Good */
(x + y) * (z + a)
```

Spaces (and indentation) do not achieve the effect of parentheses. The former of these two expressions is interpreted as \( x + (y * z) + a \), which does not appear to be what was intended.

- **Use braces to help indentation.** Automated indentation algorithms are often assisted by braces. Consider the following:

```reasonml
let x = "Long line..." ++
      "Another long line.");
let x = {"Long line..." ++
      "Another long line."});
```

The latter informs an editor that the long line spills over onto another long line, so that the editor can indent it properly.

### 3.4 Spacing

*The space bar is your friend!* Don’t be afraid to press it. It is a nice big key, so it is easy to find. Use it.

1. Surround infix operators by spaces. Write this `hd, ...tl`, not this `hd, ...tl`, and `x + y`, not `x+y`.

2. Insert spaces after the `:` in type annotation: e.g., `(17: int)`.

3. Insert one space after a pipe in a type definition.

```reasonml
type season =
| Fall
| Winter
| Spring
| Summer;
```

On the other hand, just as you should not use too many parentheses, you should not insert too many spaces! For example, do not surround a procedure’s arguments by spaces.
/* Bad */
my_procedure ( arg_1, arg_2 )

/* Good */
my_procedure(arg_1, arg_2)

4 Pattern Matching

- Pattern matching should always follow the structure of the data. Suppose you define a variant type, such as:

```reasonml
type train_car =
  | Engine
  | Boxcar(int) /* capacity */
  | Caboose;
```

When you pattern match on data of type `train_car`, your `switch` expression should follow the structure of the data in the same order, like this:

```reasonml
switch (train) {
  | Engine => ...
  | Boxcar(n) => ...
  | Caboose => ...
};
```

Also, your pattern matching should be exhaustive. This is not complete:

```reasonml
switch (train) {
  | Engine => ...
  | Caboose => ...
};
```

If your pattern matching is incomplete, the ReasonML compiler will issue a warning, as follows: “You forgot to handle a possible case here, for example...” Treat such warnings as bugs!

In some cases the compiler will flag a switch as incomplete when actually it isn’t. This is because the compiler is not smart enough to infer that all possible cases have been covered. For example:

```reasonml
switch (t) {
  | Leaf => Node (Leaf, datum, Leaf)
  | Node (left, x, right) when (datum < x) => Node (insert datum left, x, right)
  | Node (left, x, right) when (datum > x) => Node (left, x, insert datum right)
  | Node (left, x, right) when (datum = x) => Node (left, x, right)
};
```

The incorrect way to eliminate this warning would be to simply add a `catch-all` case, as follows:
Here, EMF stands for “Exhaustive Match Failure”.

The correct way to eliminate an compiler warning that arises from a failure to pattern match exhaustively is to add an explicit unguarded match as follows:

```
Leaf => Node (Leaf, datum, Leaf)
Node (left, x, right) when (datum < x) => Node (insert datum left, x, right)
Node (left, x, right) when (datum > x) => Node (left, x, insert datum right)
Node (left, x, right) when (datum = x) => Node (left, x, right)
Node (left, x, right) => failwith "EMF" /* Good */
```

This latter approach is preferable because it preserves the ability of the compiler to flag unmatched cases, which is one of the key features of ReasonML.

In summary, never appease the compiler by inserting a “catch-all”. Doing so negates the power of the compiler, and will impede your power to debug and extend your code.

- **Use pattern matching for selection.** Instead of using `fst` and `snd` to deconstruct a tuple, use pattern matching. For example:

```
type posn = (float, float);

/* Bad */
let p = some_posn;
let x = fst(p);
let y = snd(p);
x +. y;

/* Good */
let (x, y) = some_posn;
x +. y;
```

Similarly, records should be deconstructed using pattern matching:

```
type circle = {center : posn, radius : float};

/* Bad */
let circ = some_circle;
let c = circ.center;
let r = circ.radius;
let x = fst(c);
```

```
You should also steer away from using `List.hd` and `List.tl` in favor of pattern matching.

- **Pattern match using as few switch expressions as possible.** Rather than nest switch expressions, you can often pattern match against a tuple.

```reasonml
/* Bad */
switch (month) {
  | Jan => switch (day) {
    | 1  => "Happy New Year"
    | _  => ""
  ;
  | Mar => switch (day) {
    | 14 => "Happy Pi Day"
    | _  => ""
  ;
  | Oct => switch (day) {
    | 10 => "Happy Metric Day"
    | _  => ""
  ;
};
/* Good */
switch (month, day) {
  | (Jan, 1) => "Happy New Year"
  | (Mar, 14) => "Happy Pi Day"
  | (Oct, 10) => "Happy Metric Day"
  | _ => ""
};
```

- **Never use only one pipe in a switch expression.** There is never a need for only one pipe in a switch expression. In such cases, prefer `let`.

```reasonml
/* Bad */
switch (card) {
  | Hearts(n) => ...
};
/* Good */
let Hearts(n) = card;
```

- **Pattern match a procedure’s formal arguments when possible.** In ReasonML, compound types are deconstructed via pattern matching, using switch expressions, let expressions, or by deconstructing a procedure’s formal arguments. You should use the latter option if you need only the constituents of a procedure’s formal arguments, and have no use for the arguments as a whole. In this example, the procedure’s formal arguments are tuples:

```reasonml
/* Bad */

let f = (arg1, arg2) => {
  let x = fst(arg1);
  let y = snd(arg1);
  let z = fst(arg2);
};

/* Good */
let f = ((x, y), (z, _)) => {
  ...
};

There is no need to name arg1 and arg2 here, since all that is needed are their constituents. Likewise, in this example, where the procedure’s formal arguments are records:

/*@ Bad */
let f = (arg1, arg2) => {
  let x = arg1.field1;
  let y = arg1.field2;
  let z = arg2.field3;
  ...
};

/*@ Good */
let f = ({field1 = x; field2 = y}, {field3 = z; field4 = _}) => {
  ...
};

5 Verbosity

• Simplify if expressions. There are a number of equivalent ways to express the same conditional logic. In almost all cases, shorter expressions are preferred:

<table>
<thead>
<tr>
<th>Verbose</th>
<th>Concise</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (expr){true} else {false}</td>
<td>expr</td>
</tr>
<tr>
<td>if (expr){false} else {true}</td>
<td>!expr</td>
</tr>
<tr>
<td>if (expr){expr} else {false}</td>
<td>expr</td>
</tr>
<tr>
<td>if (!expr){x} else {y}</td>
<td>if (expr){y} else {x}</td>
</tr>
<tr>
<td>if (x){true} else {y}</td>
<td>x</td>
</tr>
<tr>
<td>if (x){y} else {false}</td>
<td>x &amp;&amp; y</td>
</tr>
<tr>
<td>if (x){false} else {y}</td>
<td>x !&amp;&amp; y</td>
</tr>
<tr>
<td>if (x){y} else {true}</td>
<td>x !</td>
</tr>
</tbody>
</table>

When an if expression is used for argument selection, it can be embedded within a procedure application to improve readability, as follows:

/*@ Duplication of f a b applications */
if (c) {f(a, b, x)}
else {f(a, b, y)};
/* Can be eliminated by embedding the if */
f(a, b, (if (c) (x) else (y)));

- **Don’t rewrap procedures.** When applying a procedure to another procedure, don’t rewrap the procedure if it already does what you need it to do. Here are two examples:

  /* Verbose */
  List.map((x => sqrt(x)), [1.0, 4.0, 9.0, 16.0]);
  
  /* Concise */
  List.map(sqrt, [1.0, 4.0, 9.0, 16.0]);

  /* Verbose */
  List.fold_left(((x, y) => x + y), 0);
  
  /* Concise */
  List.fold_left((+), 0);

- **Don’t misuse `switch` expressions.** Do not use `switch` when you mean `if`!

  /* Bad */
  switch (expr) {
  | true => x
  | false => y
  };

  /* Good */
  if (expr) {x} else {y};

  /* Bad */
  switch (expr) {
  | c => x /* c is a constant */
  | _ => y
  };

  /* Good */
  if (e == c) {x} else {y};

- **Don’t overuse `switch` expressions.** Do not bind expressions unnecessarily. For example, do not use `switch` to pattern match when `let` is enough:

  /* Bad */
  let x = {
    switch (expr) {
      | (y, z) => y
    }
  };

  /* Good */
  let (x, _) = expr;
• **Don’t reinvent the wheel.** Built in to the ReasonML library are a great number of ready-made procedures and data structures. You should use them, unless of course an assignment expressly forbids it!

For example, when writing a procedure that recursively walks down a list, think `fold`! Some other data structures have similar folding procedures; use them when they are available.

6 Acknowledgments

Much of this style guide is loosely based off of the CIS120 OCaml style guide at the University of Pennsylvania, which in turn copied much of its content of the CS312 OCaml style guide at Cornell University. Our content has been further adapted to ReasonML.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS 17 document by filling out the anonymous feedback form: [http://cs.brown.edu/courses/csci0170/feedback](http://cs.brown.edu/courses/csci0170/feedback)