Syntax Cheatsheet

We've worked very hard to make Reason look like JS while preserving OCaml's great semantics & types. Hope you enjoy it!

`let` binding

JavaScript	Reason
`const x = 5;`	`let x = 5;`
`var x = y;`	No equivalent (thankfully)
`let x = 5; x = x + 1;`	`let x = ref(5); x := x^ + 1;`

String & Char

JavaScript	Reason
`"Hello world!"`	Same
`'Hello world!'`	Strings must use `"`
Characters are strings	`'a'`
`"hello " + "world"`	`"hello " ++ "world"`
`"Uńĩçöðe"`	`{js|Uńĩçöðe|js}`

Boolean

JavaScript	Reason
`true`, `false`	`true`, `false`
`!true`	Same
`||`, &&, <=, >=, <, >	Same
`a === b`, `a !== b`	Same
No deep equality (recursive compare)	`a == b`, `a != b`
`a == b`	No equality with implicit casting (thankfully)

Number

JavaScript	Reason
`3`	Same *
`3.1415`	Same
`3 + 4`	Same
`3.0 + 4.5`	`3.0 +. 4.5`
`5 % 3`	`5 mod 3`

* JS has no distinction between integer and float.

Object/Record

JavaScript	Reason
no static types	`type point = {x: int, mutable y: int}`
`{x: 30, y: 20}`	Same *
`point.x`	Same
`point.y = 30;`	Same
`{...point, x: 30}`	Same

* This is the Reason spiritual equivalent; it doesn't mean it compiles to JS's object! To compile to the latter, see https://reasonml.github.io/docs/en/object.html#tip-tricks.

Array

JavaScript	Reason
`[1, 2, 3]`	`[|1, 2, 3|]`
`myArray[1] = 10`	Same
`[1, "Bob", true]` *	`(1, "Bob", true)`
No immutable list	`[1, 2, 3]`

* We can simulate tuples in JavaScript with arrays, because JavaScript arrays can contain multiple types of elements.

Null

JavaScript	Reason
`null`, `undefined`	`None` *

* Again, only a spiritual equivalent; Reason doesn't have nulls, nor null bugs! But it does have an option type (https://reasonml.github.io/docs/en/newcomer-examples.html#using-the-option-type) for when you actually need nullability.

Function

JavaScript	Reason
`arg => retVal`	`(arg) => retVal`
`function named(arg) {...}`	`let named = (arg) => ...`
`const f = function(arg) {...}`	`let f = (arg) => ...`
`add(4, add(5, 6))`	Same

Blocks

JavaScript	Reason

`const myFun = (x, y) => {`
`  const doubleX = x + x;`
`  const doubleY = y + y;`
`  return doubleX + doubleY`
`};`
	`let myFun = (x, y) => {`
	`  let doubleX = x + x;`
	`  let doubleY = y + y;`
	`  doubleX + doubleY`
	`};`

Currying

JavaScript	Reason
`let add = a => b => a + b`	`let add = (a, b) => a + b`

Both JavaScript and Reason support currying, but Reason currying is **built-in and optimized to avoid intermediate function allocation & calls**, whenever possible.

If-else (Conditionals)

JavaScript	Reason
`if (a) {b} else {c}`	Same *
`a ? b : c`	Same
`switch`	`switch` but super-powered!**

* Reason conditionals are always expressions!

** https://reasonml.github.io/docs/en/pattern-matching.html

Destructuring

JavaScript	Reason
`const {a, b} = data`	`let {a, b} = data`
`const [a, b] = data`	`let [|a, b|] = data` *
`const {a: aa, b: bb} = data`	`let {a: aa, b: bb} = data`

* Gives good compiler warning that `data` might not be of length 2. Switch to pattern-matching instead.

Loop

JavaScript	Reason
`for (let i = 0; i <= 10; i++) {...}`	`for (i in 0 to 10) {...}`
`for (let i = 10; i >= 0; i--) {...}`	`for (i in 10 downto 0) {...}`
`while (true) {...}`	Same

JSX

JavaScript	Reason
`<Foo bar=1 baz="hi" onClick={bla} />`	Same
`<Foo bar=bar />`	`<Foo bar />*`
`<input checked />`	`<input checked=true />`
No children spread	`<Foo>...children</Foo>`

* Argument punning!

Exception

JavaScript	Reason
`throw new SomeError(...)`	`raise(SomeError(...))`
`try {a} catch (Err) {...} finally {...}`	`try (a) { | Err => ...}*`

* No finally.

Blocks

In Reason, "sequence expressions" are created with `{}` and evaluate to their last statement. In JavaScript, this can be simulated via an immediately-invoked function expression (since function bodies have their own local scope).

JavaScript	Reason

`let res = (function() {`
`  const x = 23;`
`  const y = 34;`
`  return x + y;`
`})();`
	`let res = {`
	`  let x = 23;`
	`  let y = 34;`
	`  x + y`
	`};`

Comments

JavaScript	Reason
`/* Comment */`	Same
`// Line comment`	Coming soon