Caterwaul By Example

Spencer Tipping

August 14, 2011

Contents

Chapter 1

List of Primes

It's rare that you actually need to generate a list of primes in production code, but this problem makes a good programming language example. The goal here is to write a web page that ends up showing the user a list of primes below 10,000.

1.1 Preparing the HTML

The first step, assuming that we want to iterate rapidly, is to create a basic HTML document that loads Caterwaul and our application source code:

Listing 1.1 examples/primes/index.html

```
<!doctype html>
<html>
  <head>
  <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.js'></script>
  <script src='http://caterwauljs.org/build/caterwaul.js'></script>
  <script src='http://caterwauljs.org/build/caterwaul.std.js'></script>
  <script src='primes.js'></script>
  </head>
  <body></body>
</html>
```

Caterwaul's core and the `std` module don't depend on jQuery. I'm just including it here so that we can easily do things when the page is loaded.

1.2 Invoking Caterwaul

Caterwaul is a Javascript compiler written in Javascript. So to use it, you just hand your function to Caterwaul and run the function it gives you back. Here's an example of this behavior:

```
// Use an unconfigured caterwaul compiler; this has no effect:
var f = function (x) {return x + 1};
var g = caterwaul(f);

// Use all known Javascript extensions:
var f2 = function (x) {return y, where[y = x + 1]};
var c  = caterwaul.js_all();      // Returns a configured compiler
var g2 = c(f2);
```

Most Javascript apps have a surrounding lexical closure to create a local scope, so you can save some space by transforming the function inline:

```
caterwaul.js_all()(function () {
  // app code
})();

// If using jQuery:
$(caterwaul.js_all()(function () {
  // app code
}));
```

We're using jQuery, so our app will look like the second function here.

1.3 Application code

Here's a Caterwaul app that computes our list of primes and shows it to the user:

Listing 1.2 examples/primes/primes-first.js

```
$(caterwaul.js_all()(function () {
  var is_prime = function (x) {
    for (var i = 2; i * i <= x; ++i)
      if (x % i === 0)
        return false;
    return true;
  };

  var list = [];
  for (var i = 2; i < 10000; ++i)
    if (is_prime(i))
      list.push(i);

  $('body').text(list.join(', '));
}));
```

Caterwaul is a superset of Javascript, so this example behaves exactly as we'd expect. However, it doesn't look particularly special. Using Caterwaul idioms and refactoring a bit, we get this:[1]

Listing 1.3 examples/primes/primes.js

```
$(caterwaul.js_all()(function () {
  $('body').append(primes.join(', '))
  -where [
    composite(n) = ni[2, Math.sqrt(n)] |[n % x === 0] |seq,
    primes       = n[2, 10000] %!composite -seq];
}));
```

This code probably looks like voodoo, but it's actually not that complicated. There are two things happening here that aren't in regular Javascript. First, the `where[]` construct binds variables within an expression. Second, the `seq` modifier is deeply mysterious and somehow condenses five or six lines of code into two.

`where[]` is used when you want to locally bind something. For example:

```
alert(x)     -where [x = 10];
alert(f(10)) -where [f(x) = x + 1];
```

This is translated into a local function scope that gets called immediately:

```
(function () {
  var x = 10;
  return alert(x);
})();

(function () {
  var f = function (x) {return x + 1};
  return alert(f(10));
})();
```

You'll notice that there's a little bit of magic going on to let you say `f(x) = x + 1`. This is not explicitly handled by `where[]`; instead, Caterwaul has a macro that rewrites things that look like `x(y) = z` into `x = function (y) {return z}`. Because this rule is applied globally, you can also use it to create methods or reassign existing functions:

```
jQuery.fn.size() = this.length;
```

Because it's recursive, you can create curried functions by providing multiple argument lists:

[1] Thanks to Jeff Simpson for pointing out a bug here! I had previously written `composite(n) = n[2, Math.sqrt(n)]`, which has an off-by-one error since it stops before reaching $\sqrt{n}$. It's necessary to use an inclusive upper bound to make sure that we get squares as well as other factors.

```
alert(f(1)(2)) -where [f(x)(y) = x + y];

// Is compiled to:
(function () {
  var f = function (x) {
    return function (y) {
      return x + y;
    };
  };
  return alert(f(1)(2));
})();
```

A reasonable question is how `where[]` knows how much code should be able to see the variables you're binding. The answer has to do with its prefix: `where` is what's known as a modifier, and all modifiers have an accompanying operator that has a precedence. For example, in the expression `x * y -where[...]` the `where` clause binds over the multiplication, since the minus has lower precedence. Writing it as `x + y /where[...]` causes only `y` to have access to the `where` variables.

`seq` is the macro that is more interesting in this prime-generator example. It's a mechanism that writes all different kinds of `for` loops. In this case we're using it in two places. The first time is in the `composite()` function, where we use it to detect factors:

```
composite(n) = ni[2, Math.sqrt(n)] |[n % x === 0] |seq
```

First, we use `ni[]` to generate an array of numbers between 2 and `Math.sqrt(n)`, inclusive. `ni[]` (with square brackets) is a syntax form, not a function; so it won't collide with the variable `n` that we take as a parameter. The next piece, `|[n % x === 0]`, has two parts. The pipe operator, which normally performs a bitwise-or, means "there exists" in sequence context. Its body, `n % x === 0`, is then evaluated for each element (`seq` calls the element `x`). So at this point we're asking, "does there exist an integer `x` for which `n % x` is zero?" We tack the `|seq` onto the end to cause the preceding expression (in this case, everything back to the =) to be interpreted by the `seq` macro. `seq` is a modifier just like `where[]`, though `seq` doesn't take parameters.

The other use of `seq` is to retrieve all numbers that are not composite:

```
primes = n[2, 10000] %!composite -seq
```

`n[]` is doing the same thing as before. After it is `%`, which is the filter operator. So we're filtering down to only the elements which are not composite. The filter prefix is `%`, and the `!` modifier negates the condition. Because there's already a function defined, I use that instead of writing out a block. I'm using a higher-precedence prefix for `seq` as a matter of convention; I default to using a minus unless there's a reason to use something else.

Chapter 2

This is Hideous!

```
o %k%~!f %v*!~[x /!g] -seq
```
– valid Caterwaul code

A fair point. I certainly didn't design Caterwaul to look nice.[1]

2.1 Linear edits

Seek time is a large burden for programmers, just as it is for mechanical hard drives. Programmers are extremely productive when they are able to type a continuous stream of text without jumping around. For instance, consider the amount of programmer-effort required to implement this edit:

```
// Update this function to log the square of the distance:
var distance = function (x1, x2, y1, y2) {
  var dx = x1 - x2;
  var dy = y1 - y2;
  return Math.sqrt(dx*dx + dy*dy);
};
```

This is a moderate-effort change. The expression `dx*dx + dy*dy` must be factored into a variable, then the variable must be typed twice, and logging code must be added. The steps probably look about like this:

```
return Math.sqrt(dx*dx + dy*dy);
                 ^-----------^    <- select this region and cut

var d = <paste>;                  <- type this, with a paste

console.log(d);                   <- type this on a new line

return Math.sqrt(d);              <- navigate to sqrt() and type 'd'
```

[1] Otherwise it would be called something besides Caterwaul.

If you time yourself making this edit, I'm guessing it will take on the order of five or ten seconds. I doubt that it would ever become a one or two second edit even with practice. Now consider what this edit looks like using Caterwaul:

```
return Math.sqrt(dx*dx + dy*dy);
                               ^   <- move insertion point here

-se- console.log(it)               <- type this
```

This edit can easily be made in about two seconds because it is mostly linear. This is really important! It isn't just about saving a few seconds here and there; every jump and every keystroke is a potential point of failure within the edit process. If, for example, you had ended up making this edit the first time (I've done things like this on many occasions without thinking):

```
var distance = function (x1, x2, y1, y2) {
  var dx = x1 - x2;
  var dy = y1 - y2;
  var d = dx*dx + dy*dy;
  console.log(d);
  return Math.sqrt(dx);
};
```

You might then be wondering why the function was returning bad results despite logging the right thing. It could easily turn into a 60 or 120-second bug-hunt. This is the kind of thing that saps productivity and demoralizes programmers, and this is exactly what Caterwaul was designed to overcome.

2.1.1 How Caterwaul does this

I thought a lot about this problem before deciding on the current notation. Infix operators are obviously a step in the right direction (the worst offender is Lisp, which requires at least one jump for every significant edit). These operators are organized by common-use precedence, since it was far more common to add products than to multiply sums, for instance. In a sense, this is a very ergonomic Huffman-style coding of common practice.

Perhaps a less commonly recognized feature of infix operators is that you can easily determine how much code to grab with one continuous edit. For example, support I'm editing the code below and want to conditionalize the side effect on truthiness:

```
foo() -se- console.log(it)
      -re- it.bar()
```

I don't want to use short-circuit logic, since this has far lower precedence than minus. I also don't want to use a minus, since it left-associates (meaning that any modifier I used would grab everything back to the `foo()` invocation). Rather, I need a / prefix, which will modify only the log expression:

```
foo() -se- console.log(it) /when.it
      -re- it.bar()
```

The alternative to choosing an operator is to add explicit grouping. This, however, requires a jump and decreases readability, which I complain about in more detail in the next section:

```
foo() -se- (it && console.log(it))
           ^-----^               ^      <- two edit points
```

2.2 Silly parentheses

```
...  get)) add) button))))))))))) 
```
– Lisp

Relying on operator precedence for grouping has another beneficial side effect: It reduces the infamous Lisp problem of trailing parentheses.

Parentheses and other grouping constructs are hard for humans to parse unless they have some other kind of reinforcement such as indentation or differentiated brackets. For example, this code requires a lot of effort to understand:

```
sum(product(join(x, y)), fix(z(t), z()), a(b(c), d, e))
```

I think this has something to do with our natural language being primarily infix: subject-verb-object. Infix is simple because there is an implicit limit to how much stuff you can group together. If you're adding stuff, all you have to worry about is multiplication and more addition. You don't have to count parentheses to keep track of when an expression ends.

So while parentheses and other explicit grouping constructs are more versatile and computationally pure, they can impose a lot of conceptual overhead. This, along with the ergonomic advantage of infix operators, was a large influence on Caterwaul's macro design.

2.3 Keystrokes

Some things are easier to type than others. For example, it's far easier to type a dot than it is to type a caret. Part of it is frequency of use, but another part of it is layout. Caterwaul is optimized to avoid using the shift key for common cases, since this is a large cause of typos (for me, anyway). This is why it uses / and - instead of * and +.

This is also why all of its identifiers use lowercase and generally avoid underscores. The shift key produces its own set of typos, including:

```
aNameWithcaps            <- missed shift
aNAmeWithCaps            <- doubled shift
```

There's another issue though. The shift key generally occupies the hand opposite of the one typing the character. So, for instance, typing a + or a * would use the right index/middle or ring/pinky along with the left pinky on the shift. However, most Caterwaul modifiers begin with a letter typed by the left hand:

```
where
se
re
seq
when
raise
rescue
given
delay
```

So it's useful to have the left hand available for the character immediately following the modifier operator. In this case, that means using unshifted variants of + and *, which, naturally enough, are - and /.

2.4 Aesthetics

```
log(factorial(5)),
where [factorial(x) = x ? x * factorial(x - 1) : 1]
```

– not the ugliest code out there

Caterwaul won't win any beauty contests, but I've tried to make design choices that keep it visually out of the way. This is another reason I chose -, /, and | as modifier prefixes: they are all straight lines and don't introduce intersections into a stream of text.[2]

Also, Caterwaul gives you the flexibility to write code that looks very nice. Consider this numerical integration function, for instance:

```
integrate(f, a, b, h) = sum(moments)
                        -where [sum(xs) = xs /[x + x0] -seq,
                                moments = n[a, b, h] *[f(x) * h] -seq]
```

The definition of integration is clearly visible, and the implementation details follow as an afterthought. Most Javascript code requires that the implementation details come first, followed by the high-level description at the end:

```
function integrate(f, a, b, h) {
  var sum = 0;
  for (var x = a; x < b; x += h)
```

[2] I have no idea whether this actually matters, but I noticed that I preferred operators with simple linear forms to operators with more complex forms.

```
    sum += f(x) * h;
  return sum;
}
```

This imperative definition is simpler, but at the cost of eliding the `moments` variable. We can do the same with the Caterwaul definition, though it loses some readability:

```
integrate(f, a, b, h) = n[a, b, h] /[0][x0 + f(x) * h] -seq
```

Chapter 3

Game of Life

This example is more interesting than the prime list because it makes use of Caterwaul's jQuery macros. These macros, along with the sequence library, can significantly change the way Javascript programs are structured. As a twist, this game of life is implemented on a toroidal map; that is, both the horizontal and vertical edges wrap around to the opposite side.

3.1 HTML structure

Like other Caterwaul applications, this one is built on a minimal HTML file that pulls in various scripts.

Listing 3.1 examples/game-of-life/index.html

```
<!doctype html>
<html>
  <head>
  <script src='http://ajax.googleapis.com/ajax/libs/jquery/1.6.2/jquery.js'></script>
  <script src='http://caterwauljs.org/build/caterwaul.js'></script>
  <script src='http://caterwauljs.org/build/caterwaul.std.js'></script>
  <script src='http://caterwauljs.org/build/caterwaul.ui.js'></script>
  <script src='life.js'></script>
  <link rel='stylesheet' href='style.css'/>
  </head>
  <body></body>
</html>
```

Listing 3.2 examples/game-of-life/style.css

```
.cell          {background: #eee}
.cell:hover    {background: #ccc}
.cell.on       {background: #444}
.cell.on:hover {background: #555}
```

```

.board {position: relative}
```

3.2 UI structure

One of the tricky things about the game of life is linking the cells together. This is especially true when, as is the case here, we use the DOM as a data model (thus removing the usual layer of indirection that causes all sorts of synchronization bugs).

The key in this case is to build up arrays of elements first, establishing their visual position as a separate step. This avoids tricky reliance on DOM-level navigation and sibling checking. There are a couple of ways to represent the elements. One is to use a single long array and use modular access, but I'm going to use multiple nested arrays to illustrate some less commonly used features of the sequence library.

Above the board, which is contained within a single , is a button to step the simulation.

3.3 Javascript code

Normally I write Caterwaul with 192-column wrapping. This makes it easier to write multiple layers of `where[]` modifiers without running out of horizontal space. (This, in turn, keeps the left margin relatively clean by moving lower-level details off to the right.) However, this code is wrapped at under 100 columns to be readable in the PDF.

Listing 3.3 examples/game-of-life/life.js

```
$(caterwaul.jquery(caterwaul.js_all())(function () {
  $('body').append(top_button_row, board.div)
  -where [
    board           = life_board(30, 30),
    interval        = null,
    run_or_stop()   = interval ? stop_running()  -se- $(this).text('Run') :
                                 start_running() -se- $(this).text('Stop'),
    start_running() = interval = setInterval(board.step, 100),
    stop_running()  = clearInterval(interval) -se [interval = null],
    top_button_row  = jquery in div(
                        button.step('Step') /click(board.step),
                        button.run('Run')   /click(run_or_stop))]
  -where [
    life_board(x, y)    = {cells: cells,
                           div:   div_for(cells),
                           step:  step_function(cells)}
                          -where [cells = cells_for(x, y)],
```

```
18
19      cells_for(x, y)     = n[x] *~[n[y] *y[cell_for(x, y)]] -seq,
20      cell_for(x, y)      = jquery in div.cell *!x(x) *!y(y) %position(x, y)
21                                                             %invert_on_click,
22      position(x, y)(e)   = e.css({position: 'absolute',
23                                   left: x * 12, width:  10,
24                                   top:  y * 12, height: 10}),
25
26      invert_on_click(e)  = e.mousedown($(this).toggleClass('on') -given.e),
27      div_for(cs)         = jquery [div.board]
28                              -se- cs *!~[x *![it.append(x)]] /seq,
29      step_function(cs)() =
30        cs *!~[x *!update] -seq
31        -where [
32          new_state(x, y) = on(x, y) ? count(x, y) -re [it >= 2 && it <= 3] :
33                                       count(x, y) === 3,
34          count(x, y)     = adjacent(x, y) /[x + x0] -seq,
35          adjacent(x, y)  = (ni[x - 1, x + 1] - ni[y - 1, y + 1])
36                              %p[p[0] !== x || p[1] !== y]
37                              *[+on(x[0], x[1])] -seq,
38          cell(x, y)      = cs[wrap(x, cs)] -re- it[wrap(y, it)],
39          wrap(x, xs)     = (x + xs.length) % xs.length,
40          on(x, y)        = cell(x, y).hasClass('on'),
41
42          new_states      = cs *~[x *y[new_state(xi, yi)]] -seq,
43          update(cell)    = cell.toggleClass('on',
44                              new_states[cell.data('x')][cell.data('y')])]]}));
```

3.4 What's going on here?

This code is reasonably well front-loaded,[1] so I'll explain the code from the top down.

First off, there are two separate `where[]` blocks operating at the top level. This is totally ok because all of the operators used to modify things are left-associative. This means that the syntax tree will end up parsing like this:

```
($('body').append(...) -where [...]) -where [...]
```

This wouldn't be possible if - were right-associative; in that case, you'd have:

```
$('body').append(...) -(where [...] -where [...])
```

[1] Meaning that higher-level things precede lower-level ones.

3.4.1 UI code

The first part sets up the UI and appends it to the body:

```
$('body').append(top_button_row, board.div)
-where [
  board           = life_board(30, 30),
  interval        = null,
  run_or_stop()   = interval ? stop_running()  -se- $(this).text('Run') :
                               start_running() -se- $(this).text('Stop'),
  start_running() = interval = setInterval(board.step, 100),
  stop_running()  = clearInterval(interval) -se [interval = null],
  top_button_row  = jquery in div(
                      button.step('Step') /click(board.step),
                      button.run('Run')   /click(run_or_stop))]
```

This is all grouped into a single `where[]` block because it's all UI-related. Grouping things like this limits the amount of interaction that can happen between any two components.

An English-like description of each function and variable would be something like this:

`board` The model for the simulation. This provides a UI, step function, and nested cell arrays.

`interval` The interval timer used to run the simulation. When the simulation is running, this will hold the result of `setInterval()`; otherwise it will be `null`.

`run_or_stop()` If the simulation is running, then stop it and set the button text to `Run`. Otherwise, start the simulation and set the button text to `Stop`.

`start_running()` Creates and stores an interval timer that steps the simulation every 100ms.

`stop_running()` Stops the (presumably running) interval timer and clears the `interval` variable to indicate that nothing is running.

`top_button_row` This is just a `<div>` that contains the two buttons. Each one has a click handler; the step button delegates directly to `board.step()` (which is safe because `board.step` doesn't rely on any parameters or the value of `this`), and the run button references `run_or_stop` directly.

3.4.2 Model code

The model is a little unusual in that it stores its data directly on DOM elements. I did it this way to keep the example simple; it's more difficult (though often more scalable and performant) to maintain a purely logical data structure and keep it synchronized with a view layer of DOM nodes. The exact structure used is this:

```
cells = [[cell_11, cell_12, cell_13, ..., cell_1n],
         [cell_21, cell_22, cell_23, ..., cell_2n],
         ...
         [cell_m1, cell_m2, cell_m3, ..., cell_mn]]
```

where `m` is the number of columns (!) and `n` is the number of rows. So the cells are stored in column-major order if you're looking at it from the user's point of view. I didn't have a particularly compelling reason to do it this way, other than the fact that it seems more intuitive to first iterate over `x` and then `y`. (The outer loop is over `y` when using row-major order.)

There are two major pieces of code for the model. The first is the board setup code:

```
life_board(x, y)    = {cells: cells,
                       div:   div_for(cells),
                       step:  step_function(cells)}
                      -where [cells = cells_for(x, y)],

cells_for(x, y)     = n[x] *~[n[y] *y[cell_for(x, y)]] -seq,
cell_for(x, y)      = jquery in div.cell *!x(x) *!y(y) %position(x, y)
                                                       %invert_on_click,
position(x, y)(e)   = e.css({position: 'absolute',
                             left: x * 12, width:  10,
                             top:  y * 12, height: 10}),

invert_on_click(e)  = e.mousedown($(this).toggleClass('on') -given.e),
div_for(cs)         = jquery [div.board]
                      -se- cs *!~[x *![it.append(x)]] /seq
```

Here's what each variable/function does:

`life_board(x, y)` Creates a board, which is the top-level model element. A board consists of cells, a display , and a public `step()` method that advances its state by one generation. Normally a model wouldn't expose DOM elements, but this one does because of the DOM-data coupling.

The `div` and `step` attributes aren't linked to the board object itself, but they are linked to the cell array. This is useful from an API design perspective: it lets users call the `step` method without setting the context. (Which is what we did by using the `step()` method as the click handler in the previous section.) The simplest way to create unbound functions such as `step_function` is to use currying, and this is how `step_function` is defined.

`cells_for(x, y)` This constructs a column-major nested array of cell elements. Each of these s keeps track of its state by either having or not having the `on` class. Elements with the `on` class set are drawn with a dark background; this behavior is governed by the stylesheet. (This prevents the Javascript from knowing too much about the UI presentation.)

```
n[x] *~[n[y] *y[cell_for(x, y)]] -seq
```

The sequence comprehension is a bit gnarly, but here's what each piece of it means:

`n[x]` Construct an array of numbers `[0, 1, 2, ..., x - 1]`.

`*~[...]` Map over a block, and interpret the contents of the block in sequence context (that is, as if the `-seq` modifier had been applied to the body of the block). The `*` operator signifies mapping, and the `~` indicates sequence context.

Because no variable was named, the block will receive `x`, `xi`, and `xl` as lexically-scoped variables. This inner `x` shadows the `x` passed into `cells_for()` within the context of the block.

The result of this inner block, which is an array of cells, will become a single entry in the final array returned by `cells_for()`. This outer map operation constructs the array of all columns (each column is constructed by the logic below).

`n[y]` Construct an array of numbers `[0, 1, 2, ..., y - 1]`.

`*y[...]` Map over another block, this time interpreted as a regular Javascript expression. The `y` immediately preceding the block causes the loop variables to be renamed to `y`, `yi`, and `yl` instead of the default `x`, `xi`, and `xl`. The inner `y` created for this block will shadow the one passed into `cells_for()`.

This map operation constructs a column of cells.

`cell_for(x, y)` Construct a single cell at the coordinates (x, y). Because this is the result of mapping, the cell will be added to the column array.

`-seq` This causes the entire preceding expression (in this case, everything back to the `=`, which has lower precedence than the `-` on the front of `seq`) to be treated as a sequence comprehension. This is what causes `*` to be interpreted as a map operation, `n[]` to be interpreted as an array constructor, etc.

`seq` is a modifier just like `jquery` and `where[]`, so it could be written several different ways. I generally like to use `-` because it's a medium-precedence operator that easily combines with things like `-se-` and `-re-`.

`cell_for(x, y)` Constructs and returns a single cell, which is a jQuery-wrapped `<div>`. Each cell has some data associated with it. It has two jQuery `data()` properties to store its `x` and `y` coordinates, and it tracks its on/off state using the `on` CSS class.

```
jquery in div.cell *!x(x) *!y(y) %position(x, y) %invert_on_click
```

Here's what the `jquery` modifier is doing:

`jquery in` Causes the following expression to be interpreted in jQuery context. This enables the transformations described below. `in` has the same precedence as <, >, <=, and >= and is left-associative. This means that all of the usual arithmetic operators will still be in jQuery context, but any relational or lower-precedence operators will escape.

`div.cell` In jQuery context this constructs a element with CSS class `cell`. Specifically, it expands to `jQuery("<div>").addClass("cell")`.

`*!x(x) *!y(y)` This looks a bit tricky, but there isn't anything particularly magical happening. The `*!` prefix, when interpreted by the `jquery` modifier, refers to a jQuery `data` property. So in general, things of the form `a *!b(c)` will expand to `a.data("b", c)`, and that's exactly what's happening here. In this case, the resulting code is `.data("x", x).data("y", y)`, which sets two data properties and returns the original jQuery selector.

`%position(x, y)` This is interesting. First, it calls `position()` on the current `x` and `y` coordinates, then it calls that result on the current element. In general, `a %b` is expanded into `b(a)`. Because applying `position` to `(x, y)` has higher precedence than the `%` operator, the resulting expansion is `position(x, y)(...)`, where `...` is the aforementioned element constructed by `div.cell`.

`%invert_on_click` The same thing as `%position(x, y)`, but this function isn't curried. `%` left associates, so the overall parse order is `(... %position(x, y)) %invert_on_click`. By the rule described above, then, the whole expansion would be `invert_on_click(position(x, y)(...))`. You can see this happening using the interactive shell on http://caterwauljs.org.

`position(x, y)(e)` This is one of the functions used in the jQuery comprehension in the previous step. It's responsible for visually positioning each cell on the board. I've hard-coded some numbers here; basically, each cell is 10×10 pixels and has a 2-pixel margin.

Importantly, this function returns `e` (indirectly, since jQuery's `css()` method returns the original element when used as a setter). This matters because the `jquery` modifier lets you change the return value from a `%` function. For example, suppose we had written this:

```
position(x, y)(e) = e.css({...}) -re- false
```

Then the `jquery` comprehension would end up passing `false` to `invert_on_click`, with unfortunate consequences. Generally, functions used with `%` should return the element they're given to avoid confusion.

`invert_on_click(e)` This function is generally unremarkable. The only interesting thing about it is that it uses an inline `-given` modifier, which converts its expression into a function. The `.e` on the end of `given` is a shorthand to indicate

that `e` is the function's only parameter. (If we wanted a function that took multiple parameters, we'd write something like `-given[e, x, y]`.)

Like `position(x, y)`, `invert_on_click` returns the element it modifies. That is, besides the fact that it is side-effecting, it behaves like the identity function.

3.4.3 Simulation code

This is the most complex bit of the program. It contains logic to count the neighbors of each cell, compute an array of new states, and update each cell to reflect its new state.

```
step_function(cs)() =
  cs *!~[x *!update] -seq
  -where [
    new_state(x, y) = on(x, y) ? count(x, y) -re [it >= 2 && it <= 3] :
                                 count(x, y) === 3,
    count(x, y)     = adjacent(x, y) /[x + x0] -seq,
    adjacent(x, y)  = (ni[x - 1, x + 1] - ni[y - 1, y + 1])
                        %p[p[0] !== x || p[1] !== y]
                        *[+on(x[0], x[1])] -seq,
    cell(x, y)      = cs[wrap(x, cs)] -re- it[wrap(y, it)],
    wrap(x, xs)     = (x + xs.length) % xs.length,
    on(x, y)        = cell(x, y).hasClass('on'),

    new_states      = cs *~[x *y[new_state(xi, yi)]] -seq,
    update(cell)    = cell.toggleClass('on',
                        new_states[cell.data('x')][cell.data('y')])]
```

The first thing to notice is that this function is curried; it takes a cell array called `cs`, and expects to be invoked again before doing anything. This is an easy way to store temporary data using Caterwaul. I've curried this function because it ends up being invoked as the click handler of the step button. The click handler won't receive a reference to the board, so the function will need to have one already available. In this sense the function is behaving more like a callable object, and in fact currying can be a good substitute for immutable object-oriented data structures.

The other unusual thing about `step_function` is that it has a bunch of variables defined in a sub-`where[]`. These variables are re-created on each invocation of `step_function()()`.

Like other functions, this one is front-loaded; here's a breakdown of what's happening.

`cs *!~[x *!update] -seq` This just runs through the two-dimensional array, calling `update()` on each cell . Cells know their own coordinates, so we don't need

to pass the array indexes into the `update()` function; it will obtain the coordinates directly from the cell object's `data` properties.

The sequence comprehension here consists of:

`cs *!~[...]` A foreach operation on `cs`. This invokes the block, in sequence context due to the tilde, on each element of `cs`. Each element of `cs` is itself an array. The block's result isn't stored because it is used in a foreach rather than a map operation. (For this reason, foreach has less overhead than map.)

`x *!update` This block takes `x`, which is one of the arrays in `cs`, and invokes `update` on each element. In this case, `update` appears where a block would normally go. `seq` handles this by turning it into this block: `[update(x, x0, xi, xl)]`. It's safe to use `update()` this way because it uses only its first parameter. It isn't safe to use a function such as `Array.prototype.push` in this context, however, as it will push each item you pass it (which probably isn't what you would want to do).

`-seq` Modifies the preceding expression by interpreting it in sequence context. It's ok to use `-` here because of left-associativity.

```
new_state(x, y) = on(x, y) ? count(x, y) -re [it >= 2 && it <= 3] :
                             count(x, y) === 3
```

Returns a boolean specifying whether the cell at position (x, y) will be activated in the next generation. The rules for this depend on the current state of the cell. If the cell is currently on and has 2 or 3 active neighbors, then it will remain activated. A cell that is currently off and has 3 active neighbors will become activated.

As might be expected, `on(x, y)` returns a boolean indicating whether the cell at (x, y) is presently activated.

The `-re[]` expression is being used to bind a temporary variable. This is useful when you don't want to go to the trouble of setting up a `-where[]` block or when you need to box up low-precedence operators such as `&&`. It's similar to the linguistic construct people would use when saying something like "if the neighbor count of (x, y) is at least 2 and it's at most 3, then ...", though Caterwaul's construct is a bit clunkier, requiring both uses to be referred to as `it`.

```
count(x, y) = adjacent(x, y) /[x + x0] -seq
```

Returns the number of active neighbors of the cell at (x, y). This is done by retrieving the active-status of each adjacent cell (returned by `adjacent(x, y)`), and folding over addition. A less efficient but equivalent way to do it would be to use filtering: `adjacent(x, y) %[x] -seq -re- it.length`. Conceptually, I think folding communicates the intention more clearly.

```
adjacent(x, y) = (ni[x - 1, x + 1] - ni[y - 1, y + 1])
                   %p[p[0] !== x || p[1] !== y]
                   *[+on(x[0], x[1])] -seq
```

This is a fun one, and it's where most of the heavy lifting happens. The idea is to come up with an array of numerical active-statuses, each one either a 1 (active) or a 0 (inactive). The `count()` function can then sum that array with minimal effort.

`(ni[...] - ni[...])` This constructs a single array of `[x, y]` pairings within the 3×3 rectangle around (x, y). If `x` is 4 and `y` is 10, for example, then this expression returns `[[3, 9], [3, 10], [3, 11], [4, 9], [4, 10], [4, 11], [5, 9], [5, 10], [5, 11]]`. This behavior is primarily governed by `-`, which in sequence context computes the cartesian product of its operands. (Because `-` has lower precedence than most sequence operators, it must be parenthesized as shown here.)

`%p[p[0] !== x || p[1] !== y]` Here we're removing the `[x, y]` pair representing the original cell. Each pair is called `p` to avoid shadowing `x` or `y` (both of which are required inside the block). The resulting array contains eight pairs, each one of which represents a cell that's adjacent but not the original.

`*[+on(x[0], x[1])]` Now we transform coordinate pairs into numbers. The map, hopefully, is not mysterious at all by now; but the block is a bit unusual. `on()` returns a boolean, which I'm coercing into a number using Javascript's unary + operator. The block is being interpreted in regular context, so this isn't transformed by `seq` in any way.

`-seq` Wrapping up the sequence comprehension by transforming everything back to =. The return value of this sequence comprehension is an array of eight numbers, each a 1 or a 0, describing the activation states of the cells surrounding (x, y).

```
cell(x, y) = cs[wrap(x, cs)] -re- it[wrap(y, it)]
```

Normally I wouldn't factor this into its own function, since indexing twice into `cs[]` isn't a big deal. But this life game is in toroidal space, which requires wraparound logic. So the `cell()` function has to transform the array indexes as well as dereference stuff.

I'm using `-re-` here because we need two references to `cs[wrap(x)]`. The first one, obviously, provides the cell within the column. The second one is passed into `wrap()` to provide the wrapping length. I could bind a local variable or duplicate a subexpression, but `-re-` is very short to type so I've used that instead.

```
wrap(x, xs) = (x + xs.length) % xs.length
```

This wraps the index around the beginning or end of the array and returns the new transformed index. It takes the array so that it can retrieve its length.

```
on(x, y) = cell(x, y).hasClass('on')
```

A trivial function to concisely represent the activation state of the given cell.

```
new_states = cs *~[x *y[new_state(xi, yi)]] -seq
```

This is an array like `cs` that contains the new state of each cell. We have to store these before updating the individual cell elements because otherwise we would lose information; future computations depend on the old state of each cell.

My API is a bit suboptimal for this case.[2] I end up throwing away the actual elements of `cs` and just using their indexes as positional coordinates. The first map preserves sequence context for its block to make the inner map more convenient. I rename the inner variable to have access to both `xi` and `yi` at the same time.

`new_states` is a variable, so its value is computed eagerly after the functions above it have been created. However, it has no access to `update`, which is defined below.

```
update(cell) = cell.toggleClass('on',
    new_states[cell.data('x')][cell.data('y')])
```

By the time `update()` is called, `new_states` is fully populated. All we have to do is store the new state back onto the DOM element, which happens using `toggleClass()`. As usual, jQuery makes this trivially easy.

And there you have it! That's the entirety of this 41-SLOC implementation of the game of life.

[2] Actually, a lot of the factoring here is suboptimal. But I'm resisting my OCD and leaving it this way for now.