Caterwaul By Example

Spencer Tipping

July 28, 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 because of its heavy use of algorithms. 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/extensions/std.pre.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);
```

It's actually less clunky to use Caterwaul in most cases. Most Javascript apps have a surrounding lexical closure to create a local scope; all you have to do is transform this 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:

Listing 1.3 examples/primes/primes.js

```
$(caterwaul.js_all()(function () {
  $('body').append(primes.join(', '))
  -where [
    composite(n) = n[2, Math.sqrt(n)] |[n % x === 0] |seq,
    primes       = n[2, 10000] %[! composite(x)] -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 use it to create methods:

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

You can also use it to create curried functions by providing multiple argument lists:

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

First, we use `n[]` to generate an array of numbers between 2 and `Math.sqrt(n)`. `n[]` (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(x)] -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. 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.