Caterwaul By Example

Spencer Tipping

July 30, 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/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);
```

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:

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 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:

```
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.

Chapter 2

Phone Book

Note: This section is under construction; as such, the example code doesn't work yet.

Most capable functional languages can do relatively pure things like generating lists of prime numbers, but a more difficult challenge is creating a language that can adapt to impurities imposed by interfacing with the real world. This is one place where Caterwaul can be very useful; it's relatively straightforward to write new modifiers that provide library-specific functionality. One of these modules is called `ui`, and it is included with the Caterwaul distribution.

This app is a small phone book that supports JSON serialization. The user can add new phone numbers and delete existing ones, and each phone number has a name associated with it.

Listing 2.1 `examples/phone-book/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='http://caterwauljs.org/build/extensions/ui.pre.js'></script>
  <script src='overloaded-val.js'></script>
  <script src='phone-book.js'></script>
  </head>
  <body></body>
</html>
```

2.1 Model code

I'm using the word "model" in an unusual way here. It will make more sense if I preface it by saying that we know a UI is going to exist for each person in the phone book, since it wouldn't be very useful otherwise. So this code implements a "model" by providing a low-level data interface to that UI.

There are a couple of ways to do this. One is to use functional abstractions; e.g. two functions, `personToUI()` and `personFromUI()`, that perform the conversions between DOM objects and JSON. Another is to use object-oriented abstraction by overloading jQuery's `val()` method to respond to JSON values. This second method scales better for larger applications, so I'll use it here.[1]

Listing 2.2 `examples/phone-book/overloaded-val.js`

```
caterwaul.js_all()(function () {
  var original = $.fn.val;
  $.fn.overload_val(f) = this.data('overloaded-val', f);
  $.fn.val() = (this.data('overloaded-val') || original).apply(this, arguments);
})();
```

It's simpler to compose UI elements now that we can easily overload the `val()` method.

Listing 2.3 `examples/phone-book/phone-book.js`

```
$(caterwaul.js_ui(caterwaul.js_all())(function () {
  $('body').append(phone_book()),

  $('.phone-book .create button').live('click',
    delay in phone_book.find('table tbody').append(person().val(
               {name: name.val(), phone: phone.val()}))
             -then- name.add(phone).val('')

             -where [phone_book = $(this).parents('.phone-book').first(),
                     name       = phone_book.find('.create .name'),
                     phone      = phone_book.find('.create .phone')]),

  $('.phone-book .json button.save').live('click',
    delay in textarea.val(phone_book.val())
             -where [phone_book = $(this).parents('.phone-book').first(),
                     textarea   = phone_book.find('.json textarea')]),

  $('.phone-book .json button.load').live('click',
    delay in phone_book.val(textarea.val())
             -where [phone_book = $(this).parents('.phone-book').first(),
                     textarea   = phone_book.find('.json textarea')]),

  $('.phone-book table tr.person button').live('click',
    delay in $(this).parents('.person').first().remove()
             -when- confirm('Are you sure you want to remove this person?')),
```

[1]See http://github.com/spencertipping/modus for a more complete implementation of this concept.

```

  where [
    phone_book() = phone_book_ui().overload_val(phone_book_json),
    phone_book_ui() = jquery in div.phone_book(
                        table(tbody(tr(th('Name'), th('Phone')))),
                        div.create(input.name, input.phone, button('Add')),
                        div.json(textarea, button.save('Save'), button.load('Load'))),

    phone_book_json(json) =
      arguments.length ?
        this.find('table tr.person').remove()
          -then- JSON.parse(json) *[person().val(x)] *![this.append(x)] /seq
          -returning- this :
        JSON.stringify(this.find('table tr.person') *[$(x).val()] -seq),

    person() = person_ui().overload_val(person_json),
    person_ui() = jquery [tr.person(td.name, td.phone, td(button('X')))],

    person_json(person) =
      arguments.length ?
        this -effect- this.find('.name') .text(person.name)
             -effect- this.find('.phone').text(person.phone) :

        {name:  this.find('.name').text(),
         phone: this.find('.phone').text()}];
}));
```