Caterwaul By Example

Spencer Tipping

August 13, 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
```