Caterwaul in Ten Minutes

Spencer Tipping

March 4, 2013

1 Purpose of Caterwaul

Caterwaul exists to make Javascript programming easier. It is a compiler, but it is written in pure Javascript and can interoperate seamlessly with Javascript code. The simplest way to think about Caterwaul is as a Lisp-style macro layer that rewrites pieces of your functions.

Generally, Caterwaul programs look like this:

```
caterwaul(':all')(function () {
  // your program goes here
})();
```

This form is analogous to what you would normally have:

```
(function () {
  // ...
})();
```

The main difference is that we use the function `caterwaul(':all')` on the program before invoking it. Doing this allows Caterwaul to macroexpand your code.

1.1 String interpolation

Caterwaul supports Ruby-style string interpolation, which works for both single and double-quoted strings. For example:

```
caterwaul(':all')(function () {
  var place = 'world';
  console.log('hello #{place}!');
})();
```

1.2 Exercises

These exercises can be run online at http://caterwauljs.org.

1. Locate the shell on the right-hand side of the screen, and run the command `alert('hi')`. Notice that this expression returns `undefined`. Now interpolate `document.location.href` into a string to show an alert box containing the text, "you are at http://caterwauljs.org/".

2. Set a global variable `x` to `10`. Notice that using `var` causes the variable to disappear; this happens because the web shell runs each command inside a separate function.

3. Type the expression `x = y = z` at the prompt and look at the parse tree below. Based on the tree, is the = operator left or right associative?

2 Binding forms

Most Javascript code inside a Caterwaul program operates exactly as you would expect. For example:

```
caterwaul(':all')(function () {
  var place = 'caterwaul code';
  console.log('inside ' + place);
})();
```

However, some expressions will be rewritten. Caterwaul first parses your code according to Javascript syntax, then looks for parse subtrees with particular patterns. One of these patterns involves the word `where`:

```
caterwaul(':all')(function () {
  console.log('inside #{place}'),
  where [place = 'caterwaul code'];
})();
```

Caterwaul sees the `where [...]` and creates an inner function scope with local variable bindings. Here's the generated code (modulo formatting):

```
(function () {
  (function () {
    var place = 'caterwaul code';
    return console.log('inside ' + ('caterwaul code') + '');
  })();
})();
```

2.1 Modifiers

`where` is an example of a modifier, which is a general pattern for modifying some chunk of code. There are more forms than this, but some common forms are:

1. `stuff, modifier [args]`
2. `stuff -modifier [args]`
3. `stuff -modifier- args`
4. `stuff /modifier [args]`

Different forms modify different amounts of code. Because Caterwaul operates on parse trees, the Javascript operator used to separate `stuff` from `modifier` can have varying precedence and therefore apply to varying amounts of code. For example:

```
x + 1 -where [x = 5]   // (x + 1) - (where [x = 5])    -- this works
x + 1, where [x = 5]   // (x + 1), (where [x = 5])     -- this works
x + 1 -where- x = 5    // (((x + 1) - where) - x) = 5  -- fails
x + 1 /where [x = 5]   // x + (1 / (where [x = 5]))    -- fails
```

The last two cases here fail for different reasons. The first one fails because `where[]` can't see the `= 5`, since `=` has low precedence and is therefore an ancestor of the other nodes. Because of this, `where` is almost always written with brackets.

The second one fails because the `/` operator binds more tightly than the `-` operator, which means that `x` isn't a part of the code being modified by the `where` binding. As a result, `x` will be undeclared and the code will throw a `ReferenceError`.

`where` is generally written as `x, where [...]` or as `x -where [...]`.

2.2 Other binding forms

Not all binding forms are modifiers. Caterwaul also uses some rarely-written bits of Javascript syntax and turns into useful things. For example, you can write functions in declarative style:

```
f(x) = x + 1    // turns into f = function (x) {
                //               return x + 1;
                //             }
```

You can use this form anywhere, including inside a `where`:

```
console.log('#{prefix} #{message()}'),
where [prefix    = 'hi',
       message() = 'there']
```

You can also create nested functions:

```
add(x)(y) = x + y  // turns into add = function (x) {
                   //              return function (y) {
                   //                return x + y;
                   //              };
                   //            }
```

It's often useful to combine `where` and the function binding form. For example:

```
inc(x) = x + increment -where [increment = 1];
```

Since + and - have the same precedence and left-associate, the expression is parsed like this:

```
inc(x) = ((x + increment) - where [increment = 1]);
```

> **Warning!** Because `where` creates an inner function, the outer value of `arguments` is unavailable. To capture `arguments`, use a local variable:
>
> ```
> push(array, outer_args = arguments) =
>   apply(array, 'push', outer_args)
>   -where [apply(obj, name, stuff) = obj[name].apply(obj, stuff)];
> ```

2.3 Binding objects

You can create an object using `where`-style syntax. To do this, you use the `capture[]` modifier:

```
obj = capture[x = 5, y = 6]
// translates to:
// obj = {x: 5, y: 6}
```

Notice that due to the way this code is translated, you couldn't refer to `x` when defining `y` like you can with `where`. To deal with this, Caterwaul gives you another form, `wcapture`, which causes each definition to be bound to a local variable before returning the object:

```
obj = wcapture[x = 5, y = x + 1]
// translates to:
// obj = (function () {
//          var x = 5, y = x + 1;
//          return {x: x, y: y};
//        })()
```

The reason Caterwaul includes these forms is that it lets you use function binding syntax to create object methods:

```
obj.prototype = capture [
  plus(x) = this.value + x,
  times(x) = this.value * x];
```

You can also simply create those methods using dot-notation:

```
obj.prototype = {};
obj.prototype.plus(x) = this.x + x;
obj.prototype.times(x) = this.x * x;
```

2.4 Exercises

1. Use nested function notation to write the `compose` function. You can assume that it always takes exactly two functions, and that the inner function always takes exactly one argument.

2. Variables inside `where` can see each other. For example, `x + y, where [x = 5, y = x + 1]`. What happens when `x` and `y` are defined in the opposite order?

3. Write a function `args` that returns a string of its arguments. For example, it should return `[1, 2, 3]` when invoked as `args(1, 2, 3)`.

4. `where[]` is legal for any expression, including for function arguments. However, because the commas that separate function arguments are parsed with left association, you generally shouldn't use it inside an argument list. Instead, you should move the `where` outside of the function call parentheses. Do this with the example below to make the expression evaluate to `[1, 2]`.

   ```
   args(x, y, where [x = 1, y = 2])
   ```

3 Fun with sequences

Caterwaul contains a macro called `seq` that makes it very easy to manipulate arrays and objects. It works by overloading several operators to perform sequence tasks. In order to use this macro, you need to modify some code with the `seq` word:

```
expression -seq                    // interpret expression in seq context
seq[expression]                    // same thing

[1, 2, 3] *[x + 1] -seq            // * = map; returns [2, 3, 4]
[1, 2, 3] /[x0 + x] -seq           // / = fold left; returns 6
[1, 2, 3] /[0][x0 + x] -seq        // fold left with initial; also 6
[1, 2, 3] %[x & 1] -seq            // % = filter; returns [1, 3]
```

```
seq[[1, 2, 3] *[x + 1]]             // same as above
seq[[1, 2, 3] /[x0 + x]]
seq[[1, 2, 3] /[0][x0 + x]]
seq[[1, 2, 3] %[x & 1]]
```

Because seq operators have the same precedence, you can chain them:

```
sum_of_squares(xs) = xs *[x * x] /[x0 + x] -seq;
```

You can also get keys, values, or `[key, value]` pairs from objects, and reassemble pairs into objects. Like Javascript in general, Caterwaul makes no guarantees about the order in which keys or values are extracted from objects.

```
obj = {foo: 1, bar: 2, bif: 3};
obj /keys -seq              // returns ['foo', 'bar', 'bif']
obj /values -seq            // returns [1, 2, 3]
obj /pairs -seq             // returns [['foo', 1], ['bar', 2], ['bif', 3]]
obj /pairs /object -seq     // returns {foo: 1, bar: 2, bif: 3}

obj /keys *[x.length] -seq  // returns [3, 3, 3]
```

Notice that `/object` folds key-value pairs into an object. A variant of this, `/mobject`, folds key-value pairs into a multi-object; that is, an object that maps keys to arrays of values. For example:

```
pairs = [['foo', 1], ['bar', 2], ['foo', 3]];
pairs /mobject -seq         // returns {foo: [1, 3], bar: [2]}
```

You can use this to group things:

```
group_by(f, xs) = xs *[[f(x), x]] /mobject -seq;
```

3.1 Exercises

1. Rewrite `sum_of_squares` to square and add using a single fold step. (Hint: you need to use a fold-left-with-initial to do this.)
2. Write a function that returns the average value of an array.
3. Write a function that returns the largest value in an object.

4 Generating DOM elements

Caterwaul contains a macro that interfaces with jQuery to build DOM elements. This macro, called `jquery`, is similar to `seq` in that it changes the meaning of operators inside an expression.

The Caterwaul web shell will detect jQuery objects and render both their HTML and their DOM as results. The `jquery[]` macro always returns a jQuery object.

```
jquery[div('hello')]                // returns a div with the text "hello"
jquery[div.foo('hi')]               // <div class='foo'>hi</div>
jquery[div.foo.bar.bif]             // <div class='foo bar bif'></div>
jquery[div.foo_bar]                 // <div class='foo-bar'></div>
jquery[div(button('click me'))]     // <div><button>click me</></>

// calling jQuery methods:
jquery[div /text('hi')]             // <div>hi</div>
```

This macro becomes useful when you want to generate DOM contents from inside functions:

```
greeter_for(name) = jquery[div.person /text(name)];
$('body').append(greeter_for('spencer'));
```

You can use `jquery[]` with `seq[]`, but jQuery doesn't know what to do with arrays of jQuery objects. When generating arrays of jQueries, you should fold them into a single collection:

```
div_for(number) = jquery[div /text(number)];
wont_work = [1, 2, 3] *[div_for(x)] -seq;
will_work = [1, 2, 3] *[div_for(x)] /[x0.add(x)] -seq;
$('body').append(wont_work);        // DOM exception 8
$('body').append(will_work);        // this works
```

4.1 Exercises

1. Write a function that takes a name and email address, and returns a two-cell table row with these values. (Hint: `jquery[]` works with all types of elements.)

2. Write a function that takes an object of the form `{name1: email1, name2: email2, ...}` and returns a table containing everyone's name and email address.

5 Answers to exercises

Section 1

1. `alert('you are at #{document.location.href}')`
2. `x = 10`
3. = is right-associative. A left-associative parse tree would be `("=" ("=" x y) z)`.

Section 2

1. `compose(f, g)(x) = f(g(x))`
2. When defined in the opposite order, `y` takes the value `undefined1` because `x` has not yet been initialized. (This is the same thing that happens with `var`.)
3. ```
   args(arg = arguments) = '[#{xs.join(", ")}]'
                             -where [xs = Array.prototype.slice.call(arg)]
   ```
4. `args(x, y) -where [x = 1, y = 2]`

Section 3

1. `sum_of_squares(xs) = xs /[0][x0 + x*x] -seq`
2. `average(xs) = (xs /[x0 + x] -seq) / xs.length`
3. `largest_value(o) = o /values /[Math.max(x0, x)] -seq`

Section 4

1. ```
   row(name, email) = jquery[tr(td.name  /text(name),
                                td.email /text(email))]
   ```
2. ```
   table(people) = jquery[table /append(rows)]
     -where [row_array = people /pairs *[row(x[0], x[1])] -seq,
             rows      = row_array /[x0.add(x)] -seq]
   ```

Errata

§2, ex 3 [Hat tip Alan Liu]

The answer was incorrectly listed as:

`args() = '[#{arguments.join(", ")}]'`

This answer fails because the `arguments` object defines no `join` method (unlike arrays, which do). The fix is to convert `arguments` into an array before using it inside the string, which can be done in a few different ways:
   ```

```
args() = '[#{Array.prototype.slice.call(arguments).join(", ")}]'

args(arg = arguments) = '[#{xs.join(", ")}]'
                        -where [xs = Array.prototype.slice.call(arg)]

// using the seq[] macro:
args() = '[#{seq[+arguments].join(", ")}]'
```

The prefix + in sequence context causes an expression to be sliced into an array. This isn't documented in the *ten minutes* guide, but it is described on the Caterwaul website.