# Caterwaul Reference Manual

Spencer Tipping

December 23, 2011

# Contents

**1 The `caterwaul` global** **2**

**2 Syntax trees** **5**

2.1 Tree structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

2.2 Tree methods . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

**3 Example: a node.js source preprocessor** **10**

3.1 Writing a debugging macro . . . . . . . . . . . . . . . . . . . . . . 11

3.2 Using Caterwaul inside Caterwaul . . . . . . . . . . . . . . . . . . 13

# Chapter 1

# The `caterwaul` global

Caterwaul is two different things, at least to me. I tend to use it as a programming language because of its standard macro library. But beneath those macros, Caterwaul is a general-purpose Javascript syntax tree API. This separation is achieved by placing the ideas in separate files: `caterwaul.js` contains Caterwaul as a generic code library, and `caterwaul.std.js` and `caterwaul.ui.js` contain macros that you can enable by calling `caterwaul(’js_all jquery’)`. This guide talks exclusively about `caterwaul.js`.

Caterwaul introduces exactly one global variable called `caterwaul`. When you’re using Caterwaul as a programming language, you invoke this global on a string containing configurations; for example `caterwaul(’js_all jquery’)`. However, this is an abstraction over some more basic functions. Here are the most useful public methods of the global `caterwaul` object:

`parse(object)` Parses `object` as Javascript, and returns a syntax tree. The string representation of `object` is obtained by invoking `object.toString()`; this works for strings, functions, and other syntax trees.[1]

For example, here’s a quick way to test `parse()` (this can be run from the root directory of the Caterwaul repository):

```
$ node
> caterwaul = require('./build/caterwaul.node').caterwaul
[output]
> caterwaul.parse('x + y').structure()
'("+" x y)'
>
```

`compile(tree)` Similar to Javascript’s native `eval()`, but works on syntax trees. Unlike `eval()`, this method always returns a value. This means that the syntax

[1] Though as an optimization, Caterwaul is allowed to behave as an identity function if you send a syntax tree to `parse()`.

tree you pass to compile() must be an expression, not a statement or statement block.[2] This may seem like a tremendous limitation, but it isn't too bad since you can create anonymous functions:

```
> caterwaul.compile(caterwaul.parse(’if (true) console.log(5)’))
Error: Unexpected token if while compiling ...
> code = ’function () {if (true) console.log(5)}’;
> caterwaul.compile(caterwaul.parse(code))
[Function]
> caterwaul.compile(caterwaul.parse(code))()
5                       // from console.log()
undefined               // return value from function
>
```

compile() takes two optional arguments. The first is an object containing named references. This is useful when you want to pass state from the compile-time environment into the compiled expression. For example:

```
> tree = caterwaul.parse(’x + y’)
> caterwaul.compile(tree)
ReferenceError: x is not defined
> caterwaul.compile(tree, {x: 3, y: 4})
7
>
```

Caterwaul passes these values in by constructing a closure and evaluating your code inside of that closure scope. This means that you can pass in any value, not just ones that can be easily serialized:

```
> caterwaul.compile(tree, {x: caterwaul, y: tree})
'function () {return f.init.apply(f, arguments)}x+y'
>
```

The other optional argument to compile() (which must appear in the third position if you're using it) is an object containing compilation flags. As of version 1.1.5, the only flag supported is gensym_renaming, which defaults to true. You will probably never care about this; it causes any Caterwaul-generated symbol to be turned into a more readable name before the expression is returned.

Aside from a few utility methods like merge(), those methods are all that you're likely to care about on the Caterwaul global. In addition to those methods, Caterwaul also gives you access to two kinds of syntax trees:

[2] Expressions are valid when wrapped in parentheses; statements aren't. compile() wraps its tree in parentheses and executes that.

`caterwaul.syntax` This represents an ordinary Javascript expression that would come out of the `parse()` function. For example:

```
> caterwaul.parse('foo(bar)').constructor === caterwaul.syntax
true
> new caterwaul.syntax('()', 'foo', 'bar').toString()
'foo(bar)'
>
```

`caterwaul.syntax` is covered in more detail in the next chapter.

`caterwaul.ref` This gives you a way to insert a reference into compiled code. You can do this by passing a reference into `compile()`, but sometimes it's easier to use an anonymous reference. Here's how this works:

```
> tree = caterwaul.parse('foo(bar)')
> ref = new caterwaul.ref(function (x) {return x + 1})
> caterwaul.compile(tree.replace({foo: ref}), {bar: 5})
6
>
```

# Chapter 2

# Syntax trees

Most of Caterwaul's cool functionality is implemented as methods on syntax trees, and all of the standard macros make liberal use of these methods. They exist in several categories. First, there are a bunch of methods that help you use syntax trees as patterns or templates. For example:

```
> pattern = caterwaul.parse('_x + _y');
> match = pattern.match(caterwaul.parse('f(z) + bar'))
{ _x: {...}, _y: {...}, _: {...} }
> match._x.toString()
'f(z)'
> match._y.toString()
'bar'
> match._.toString()
'f(z) +bar'         // please forgive Caterwaul's questionable whitespace style
>
```

The exact semantics of `match()` are that it treats anything starting with an underscore as a wildcard. The result of `match()` is either `null` (or `undefined`) or an object that maps each underscore-wildcard to the subtree that matched at that position. It then maps the underscore itself to the entire matching tree; that is, `x.match(y)._ === y` for all `y`.

`match()` by itself is boring, but there's a complementary method called `replace()` that makes it awesome:

```
> new_pattern = caterwaul.parse('_x(_y) + _y');
> new_tree = new_pattern.replace(match);
> new_tree.toString()
'f(z) (bar) +bar'
>
```

## 2.1 Tree structure

A syntax tree looks like an array, except that it also has a `data` property. It is also assumed that a syntax tree won't be modified after it is constructed; almost all of the methods available for trees are nondestructive.[1]

```
> t = caterwaul.parse('foo + bar');
{ '0': { data: 'foo', length: 0 },
  '1': { data: 'bar', length: 0 },
  data: '+',
  length: 2 }
> t[0]
{ data: 'foo', length: 0 }
>
```

Most of the time you won't need to deal with this structure, as the methods below cover the most common use cases. But all of these methods ultimately interact with this array-like structure.

## 2.2 Tree methods

Here's the complete rundown of useful methods on syntax trees:[2]

`match` Typically used as `pattern.match(tree)`, and returns either an object, or `undefined` or `null`.

Explained above, `match()` is used to perform pattern matching on syntax trees. This method completes in $O(n)$ time and $O(d + k)$ GC overhead, where $n$ is the total number of nodes in the pattern tree, $d$ is the maximum depth of the pattern tree, and $k$ is the number of wildcards in the pattern tree. Entries in the resulting object are references, not copies, of the matching subtrees.

`replace` Typically used as `template.replace(match)` or `template.replace({v1: t1, v2: t2, ...})`, and returns a new syntax tree.

This method is basically the inverse of `match()`. The object passed to `replace()` dictates the replacements to perform. Generally, templates are written with underscore-prefixed variables (though you don't have to do it this way), and you then build objects that map underscore-prefixed keys to syntax trees. For example:

```
> caterwaul.parse('_x + _y').replace({_x: 'foo', _y: 'bar'}).toString()
'foo +bar'
>
```

[1] The only exceptions are `push` and `pop`, which are helpful when working with flattened trees. See `flatten` and `unflatten` in the method list for more details about this.

[2] Syntax trees also have a number of internal methods, prefixed with underscores. These are useful only for Caterwaul's parser and you shouldn't use them.

You can pass in strings (instead of trees) as values, as in the example above. If you do this, each non-tree will be promoted into a syntax tree with no children.

`toString` Typically used as `tree.toString()`.

Generates compilable, but not particularly nice-looking, Javascript code for the receiver. This method is optimized for performance by building an intermediate array and then using one `join()` call, so the GC overhead should be $O(n)$ in the total length of the serialized tree. Caterwaul uses this output when compiling functions, and it can be used to inspect code.

`structure` Typically used as `tree.structure()`.

A more detailed representation than `toString`. This method generates an S-expression that describes the structure of the parse tree. For example, `toString()` might return `3 + x * 10`, but `structure` would return `("+" 3 ("*" x 10))`.

`id` Typically used as `tree.id()`.

Returns a unique string identifying the receiver. This is useful when you need to keep track of a syntax tree, such as when maintaining a set of visited nodes using a Javascript object.

`as` Typically used as `tree.as(’+’)` or similar.

Takes the destination tree type. If the receiver is of this type, then `as()` returns the receiver; otherwise, `as()` returns a unary node of the given type whose child is the receiver.

This is useful when you want to unify several cases into a single bit of logic. For instance, suppose you’re writing a macro that allows the user to enter either an array (inside brackets), or a single item (not inside anything). You can invoke `node.as(’[’)` to convert the non-bracketed case into a single-element bracketed case. This lets you eliminate the non-bracketed case from the majority of your macro logic.

`flatten` Typically used as `tree.flatten(’*’)` or similar.

Normally binary operators are arranged into binary trees by their precedence and associativity. So, for example, `3 + 4 + 5` parses out to become `(+ (+ 3 4) 5)`. However, sometimes it’s useful to have a syntax tree that contains all elements of a summation at the same level. `flatten` constructs a flattened tree based on the associativity of the operator. So, for example:

```
> caterwaul.parse(’3 + 4 + 5’).structure()
'("+" ("+" 3 4) 5)'
> caterwaul.parse(’3 + 4 + 5’).flatten(’+’).structure()
'("+" 3 4 5)'
>
```

`flatten()` takes a string to indicate the kind of node you want to flatten over. This can be any Javascript operator. If the receiver isn't that kind of node, `flatten` returns a unary node of the type you provided that contains only the receiver. This is useful for cases like `f(x)`, which can still be flattened under + and will become a + node whose only child is `f(x)`. This means that regardless of the type of the receiver, you can always flatten it and iterate over its children with the same effect.

Nodes returned from `flatten()` can be used much like arrays; for example, this function will parse and evaluate a numeric sum:

```
> evaluate = function (sum_as_string) {
    var terms = caterwaul.parse(sum_as_string).flatten('+');
    for (var i = 0, total = 0, l = terms.length; i < l; ++i)
      total += +terms[i].data;
    return total;
  };
> evaluate('1 + 2 + 3 + 4')
10
>
```

`unflatten` Typically used as `tree.unflatten()`.

The inverse of `flatten`, with the caveat that it won't delete unary nodes that `flatten()` may have created. The receiver is converted to binary form according to the associativity of its operator, and the resulting node will contain only binary instances of the receiver's operator. If the receiver is unary or nullary, then the return value is structurally equivalent to the receiver.

`each` Typically used as `tree.each(f)`, where `f` is a function.

Invokes `f` on each direct child of `tree`, returning the receiver. `f` is actually called on two parameters. The first is the child, and the second is the child's numeric index (starting at 0).

`map` Typically used as `tree.map(f)`, where `f(child, i)` returns a new child or `false`.

Invokes `f` on each direct child of `tree`, returning a new tree with the same data as the receiver, but whose children are the return values of `f`. If `f` returns `false` for any child, the original child is used.

`reach` Typically used as `tree.reach(f)`, where `f` is a function.

Similar to `each`, except that `f` is invoked on the receiver and all of its descendants in depth-first pre-order. `f` receives only one parameter when invoked on the root node; for all other nodes it receives two.

`rmap` Typically used as `tree.rmap(f)`, where `f(node, i)` returns a new node, `true`, or `false`.

Similar to `map`, except that `f` is invoked on the receiver and all of its descendants in depth-first pre-order. A number of rules apply:

(a) If `f(node, i)` returns `node` or `false`, then children of `node` are visited.

(b) If `f(node, i)` returns `true`, then `node` is preserved and its descendants are not visited.

(c) If `f(node, i)` returns a new node, then the new node replaces `node` and its descendants are not visited.

`peach` Typically used as `tree.peach(f)`, where `f` is a function.

Semantically identical to `reach`, except that traversal happens in depth-first post-order. That is, a node is visited after, not before, its children are visited.

`pmap` Typically used as `tree.pmap(f)`, where `f(node, i)` returns a new node, `true`, or `false`.

Semantically similar to `rmap`, except that traversal happens in depth-first post-order and all descendants are always visited. (This has to be the case, since the return value of `f(node, i)` is unknown until after all descendants of `node` have been visited.)

`clone` Typically used as `tree.clone()`.

Returns a deep copy of the receiver.

`collect` Typically used as `tree.collect(predicate)`, where `predicate(node)` returns `true` or `false`.

Builds and returns an array of all descendants (possibly including the receiver) for which `predicate(node, i)` returns a truthy value. The array will be in depth-first pre-order.

`contains` Typically used as `tree.contains(predicate)`, where `predicate(node)` returns `true` or `false`.

Returns the first descendant (or the receiver) for which `predicate(node, i)` returns a truthy value, or `undefined` if `predicate` matches no trees.

# Chapter 3

# Example: a node.js source preprocessor

Now that you've seen the gory details of what Caterwaul is made of, let's put it to good use by writing a handy preprocessor for node.js. The most trivial way to preprocess things is to just parse them and convert them back to strings, so let's do that first. Here's the basic setup:

```
$ ls
caterwaul.node.js  my-preprocessor.js
$ cat my-preprocessor.js
var caterwaul = require('./caterwaul.node.js').caterwaul;
$
```

Normally you do asynchronous IO in node, but since this application doesn't need to field any HTTP requests we can afford to block. Here's a basic identity preprocessor:

Listing 3.1 `examples/nodejs-preprocessor/identity.js`

```
var caterwaul = require('./caterwaul.node.js').caterwaul;
var filename  = process.argv[2];
var source    = require('fs').readFileSync(filename, 'utf8');
var parsed    = caterwaul.parse(source);
require('fs').writeFileSync(
  filename.replace(/\.js$/, '.out.js'),
  parsed.toString(),
  'utf8');
```

We can now run this on itself to make sure it works (if you've got the Caterwaul repo checked out, it's in `doc/examples/nodejs-preprocessor`):

```
$ ls
caterwaul.node.js  identity.js  ...
```

```
$ node identity.js identity.js
$ ls
caterwaul.node.js  identity.js  identity.out.js  ...
$
```

At this point, `identity.out.js` is a less-readable version of the program above. If you run it on itself, it should produce itself again; Caterwaul's parse/serialize cycle stabilizes after a single iteration.[1]

## 3.1 Writing a debugging macro

Ok, let's make this thing useful. As a use case, let's say that you want to be able to write `/log` after any expression to side-effectfully log it but still return its usual value. Normally a similar effect can be achieved by writing a `trace` function that looks something like this:

```
var trace = function (x) {
  console.log(x);
  return x;
};
```

But being strongly anti-Lisp (and thus anti-parenthesis),[2] you want a less disruptive way to write it. So you're opting for the more Caterwaul-idiomatic operator form. Here's the basic transformation you want:

```
x /log  ->  (function (y) {console.log(y); return y})(x);
```

It doesn't matter what the function parameter is called, so we can just name it something generic like `x` or `y`. Division is also high-enough precedence that we don't need to worry about changing the meaning of a comma inside `x`.[3] There are more and less tedious ways to write this transformation. Here's one way to do it:

```
var log_template = caterwaul.parse('_x /log');
var expansion_template = caterwaul.parse(
  '(function (x) {console.log(x); return x})(_x)');
var each = function (subtree) {
  var match = log_template.match(subtree);
  return match && expansion_template.replace(match);
```

[1] I'm not absolutely sure about this, but I strongly suspect it to be the case and have never seen it fail to converge.

[2] Irony of putting anti-parenthetical sentiments inside parentheses fully intended.

[3] Imagine if we wanted to write `x, log`. Because the comma left-associates, `x, y, log` would end up rewriting into a function call on `(x, y)`, which Javascript will interpret as two separate arguments rather than as one argument which is a comma. When this danger is present, you need to wrap the argument in another set of parentheses. Fortunately, it won't happen here because / is much higher-precedence than comma.

```
};
var transform = function (tree) {
  return tree.rmap(each);
};
```

This will mostly work. There's only one significant problem, and it arises due to the way `rmap()` works. Let's suppose `each()` replaces a sub-tree. `rmap()` won't then descend into that subtree to re-expand any `/log` expressions that it contains. (For example, consider something like `f(x /log) /log`.) We can fix this by explicitly expanding the match body like this:

```
var each = function (subtree) {
  var match = log_template.match(subtree);
  return match && expansion_template.replace({_x: transform(match._x)});
};
```

Here's the complete program:

Listing 3.2 `examples/nodejs-preprocessor/debugger-rmap.js`

```
var caterwaul = require('./caterwaul.node.js').caterwaul;
var filename  = process.argv[2];
var source    = require('fs').readFileSync(filename, 'utf8');
var parsed    = caterwaul.parse(source);

var log_template = caterwaul.parse('_x /log');
var expansion_template = caterwaul.parse(
  '(function (x) {console.log(x); return x})(_x)');
var each = function (subtree) {
  var match = log_template.match(subtree);
  return match && expansion_template.replace({_x: transform(match._x)});
};
var transform = function (tree) {
  return tree.rmap(each);
};

require('fs').writeFileSync(
  filename.replace(/\.js$/, '.out.js'),
  transform(parsed).toString(),
  'utf8');
```

And a test case:

Listing 3.3 `examples/nodejs-preprocessor/debugger-test.js`

```
var f = function (x) {
  return x + 1;
};
f(5 /log, 10 /log) /log;
```

If this seems like a lot of work to go to just for one macro, that's probably because it is. A simpler way to go about it is to create a custom compiler that implements our macro for us. This gets rid of the explicit rmap and provides us with a way to indicate that we want to re-expand something. Here's what that looks like:

```
var each = function (subtree) {
  var match = log_template.match(subtree);
  return match && expansion_template.replace({_x: this(match._x)});
};
var transform = caterwaul(each);
```

Notice the strange call to this() inside each. When you hand a macroexpander function like each to Caterwaul, it invokes it on each node in the syntax tree just like rmap does. However, it also passes the current compiler in as this. This lets you easily re-expand subtrees like we're doing here. It also makes it so that we don't need an external reference to the transform() function; now we can inline that into the main logic:

Listing 3.4 examples/nodejs-preprocessor/debugger-caterwaul.js

```
var caterwaul = require('./caterwaul.node.js').caterwaul;
var filename  = process.argv[2];
var source    = require('fs').readFileSync(filename, 'utf8');
var parsed    = caterwaul.parse(source);

var log_template = caterwaul.parse('_x /log');
var expansion_template = caterwaul.parse(
  '(function (x) {console.log(x); return x})(_x)');
var each = function (subtree) {
  var match = log_template.match(subtree);
  return match && expansion_template.replace({_x: this(match._x)});
};

require('fs').writeFileSync(
  filename.replace(/\.js$/, '.out.js'),
  caterwaul(each)(parsed).toString(),
  'utf8');
```

## 3.2 Using Caterwaul inside Caterwaul

This is where things get fun. The example above can be condensed like crazy once we have the Caterwaul standard library to work with. To do this, let's first build a node.js file that contains what we need:

Listing 3.5 examples/nodejs-preprocessor/build-node-std

```
#!/bin/bash
curl caterwauljs.org/build/caterwaul.{node,std}.js > caterwaul.node.std.js
```

When this is required, we’ll have a Caterwaul instance that can be customized with js_all. First, let’s make sure this works by writing a small test:

Listing 3.6 examples/nodejs-preprocessor/caterwaul-test.js

```
var caterwaul = require(’./caterwaul.node.std.js’).caterwaul;
caterwaul(’js_all’)(function () {
  console /˜log/ ’hello from inside caterwaul!’})();
```

So far so good. Now let’s start leveraging some of the awesomeness that is the Caterwaul standard library. A couple of optimizations immediately jump out. First, we can use quotation instead of invoking caterwaul.parse directly. Quotation is both a modifier and a flag we can put onto string literals. Either will work in this case; I’ll use the modifier. Now we can write this:

```
var log_template = _x /log -qs;
var expansion_template = (function (x) {
  console.log(x);
  return x;
})(_x) -qs;
```

This is an important bit of progress. When you mark something with the qs modifier, Caterwaul parses it up-front and drops a reference to the parse tree into your code. This behavior is different from invoking caterwaul.parse, which will re-parse the same code every time you call it. This means that now we can inline the log template and the expansion template into our code:

```
var each = function (subtree) {
  var match = (_x /log -qs).match(subtree);
  ...
};
```

Using some of Caterwaul’s syntactic niceties, we can end up writing this:

```
each(subtree) = expansion_template /˜replace/ {_x: this(match._x)} /when.match
        -where [match              = qs[_x /log] /˜match/ subtree,
                expansion_template = qse[_x -se- console.log(it)]];
```

One of the cool parts here is qse, which is just like qs but quotes the tree *after Caterwaul has macroexpanded it*. This means that -se- will disappear and turn into something remarkably similar to our original expansion template.[4]

Once you’ve loaded caterwaul.std.js (which we did by appending it to caterwaul.node.js), the caterwaul global has some extra functions that are particularly useful for writing macros like this. They are pattern, expander, reexpander, replacer, and rereplacer. Here’s what each one does:

[4] The only semantic difference is that -se- invokes the function by using .call(this, x), which makes it so that any references to this inside the side-effect refer to the outer this.

`pattern` Takes a string or syntax tree and returns a function that calls its matcher. Logically, `caterwaul.pattern(x)(tree) = caterwaul.parse(x).match(caterwaul.parse(tree))`, though `pattern()` doesn't re-parse its argument every time.

`expander` Takes a string or syntax tree and returns a function that calls its replacer. Logically, `caterwaul.expander(x)(object) = caterwaul.parse(x).replace(object)`.

`reexpander` Just like `expander`, but returns a function that invokes `this` on the expansion output.

`replacer` Takes two string-or-syntax-trees and returns a function that matches its argument against the first argument and replaces using the second. It's basically like the `each` we defined earlier, but it doesn't re-expand the matching region(s).

`rereplacer` Just like `replacer`, but the output after replacement is re-expanded using `this`. This is basically identical to the semantics of `each`, so we'll use this to write our new-and-improved macroexpander.

Here's the idiomatic Caterwaul implementation of our preprocessor (I've aliased `$` to `caterwaul` for conciseness):

Listing 3.7 `examples/nodejs-preprocessor/caterwaul-implementation.js`

```
var caterwaul = require('./caterwaul.node.std.js').caterwaul;
caterwaul('js_all')(function () {
  fs.writeFileSync(output_file, $(expand)(parsed).toString(), 'utf8')
  -where [fs          = require('fs'),
          output_file = process.argv[2].replace(/\.js$/, '.out.js'),
          parsed      = $.parse(fs.readFileSync(process.argv[2], 'utf8')),
          expand      = $.rereplacer(_x /log -qs, _x -se- console.log(it) -qse)]},
  {$: caterwaul, require: require})();
```

The only particularly interesting thing is that we're binding some variables outside of the function's context. In particular, we're binding `require` to itself, which probably seems strange. I was kind of surprised the first time I found that this was necessary. It turns out that `require` isn't global in node.js; it's a lexically-scoped variable that is available at the toplevel but not from inside a globally-compiled function. So if you remove the binding for `require`, the program will fail with a `ReferenceError`.

> **An interesting aside:** If you run this program on itself it will emit something that's invalid! This is kind of an interesting point, I think. Our macroexpander doesn't know anything about Caterwaul's standard library or quotation or anything; it's just a blind replacer that looks for trees of the form `something /log`. Well, we happen to have a tree like that in the original: `_x /log -qs`. Our macroexpander will happily preprocess this into its expansion, `(function (x) {console.log(x); return x})(_x)`. We'll then end up with a program that contains something like this gem:

```
$.reexpander((function (it) {console.log(it); return it}).call(this, _x) -qs,
             _x -se- console.log(it) -qse);
```

This re-expander, of course, doesn't do anything (except that it will infinite-loop if something matches the pattern, since the pattern and the expansion are the same).