# Caterwaul Reference Manual

Spencer Tipping

December 8, 2011

# Contents

# 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.