Caterwaul Reference Manual

Spencer Tipping

December 8, 2011

Contents

Chapter 1

The `caterwaul` global

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 tree you pass to `compile()` must be an expression, not a statement or statement block.[2]

`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:

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

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

```
> 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.

`gensym([name])` Returns a guaranteed-unique symbol. If `name` is given, then the symbol will begin with `name`. For example:

```
> caterwaul.gensym('foo')
'foo_1_pWVi5y82xjbMJo3QxUTW03'
>
```

I say that this is guaranteed-unique, but technically it isn't. Caterwaul's gensyms contain 128 bits of random data as a suffix, and this won't occur anywhere in your code. However, it is not difficult to predict future values of `gensym()` given previous values, since Caterwaul's suffix doesn't change. I've never run into a case where this was a problem, but you can easily thwart `gensym()` if you deliberately try to.

`deglobalize()` Restores the original value of the global called `caterwaul`, and returns the receiver. This is useful if (1) you're using two versions of Caterwaul at the same time, or (2) in the unlikely event that someone else has also named their library Caterwaul.

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:

`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
>
```