Caterwaul language reference

Spencer Tipping

March 27, 2012

Contents

Chapter 1

Introduction

Caterwaul is a strange project. It began as a proof of concept, the idea being that you could usefully macroexpand Javascript functions from inside a runtime environment. Since then it has grown to include an offline precompiler, self-replication, and a bunch of other features that just barely work together.

Caterwaul 2.0 is a reboot. I'm taking everything I've learned from previous versions and am converting the project to something much more along the lines of CoffeeScript. In particular, my goal is to address these major issues:

1. Caterwaul-generated Javascript is unreadable and not idiomatic, which makes it impossible to debug caterwaul code.
2. Caterwaul syntax is ad-hoc and has no compile-time semantic checks.
3. The parser never managed to cover all edge cases on all browsers.
4. The low-level syntax tree interface makes it too easy to produce syntax errors.
5. Tree rewriting was optimized for ad-hoc online rewriting, which stopped making sense after `defmacro` went away.

The project hasn't lost any of the ugliness or pragmatism that it had before. If anything, it's only gotten worse. But hopefully in such a way that it's more useful.

1.1 Ergonomics and readability

Just like before, caterwaul is about ergonomics at the potential expense of readability. The motivation behind this tradeoff is that on the aggregate, good developers will write more readable code if it's easier to do so; but it's more important that the code exists than that it be readable.[1] To this end, caterwaul 2:

1. Inherits infix operator-precedence grouping from caterwaul 1.x.
2. Bakes SDoc syntax into the language itself and provides hash-marker line comments.
3. Allows for thin, Turing-complete notational abstraction, *preferring this to semantic abstraction*.

This last point is worth discussing. A lot of FP languages make it very easy to introduce semantic abstraction into programs. This, in turn, produces layers of indirection. When abstractions are well-defined, this adds value to the resulting code. But fairly often these abstractions are massaged into notation anyway and add complexity.[2]

Caterwaul prefers notational abstraction to semantic abstraction. This furthers the goal of providing good ergonomics, but it also has a broader philosophical context. When writing caterwaul 1.x code, I observed that it was not difficult to work at a low level of semantic abstraction but with better notation. For example, extensive use of the `seq` macro made it easy to perform complex data transformation without introducing extra structure into the program. This had two beneficial effects: first, it minimized indirection, which eased debugging and improved performance; and second, the resulting notation was more concise and usable than anything that could have been implemented using semantic abstraction.

Put differently, caterwaul erases abstractions at compilation time by pre-expanding them. In this sense it encourages well-defined compile-time invariants.

[1] I have no idea whether this assertion is true, but I think it is for me so I'm running with it for now.

[2] Consider the dark voodoo behind ScalaTest, for instance.

Chapter 2

Contexts

TODO: Figure out what a context is.

Chapter 3

Notational abstraction

Caterwaul is an interpreted programming language whose programs generally return Javascript syntax trees. This means that semantic abstraction within a caterwaul program is erased by the time the Javascript is rendered; therefore, it corresponds to notational abstraction in the resulting output. The ability to introduce semantic abstraction into the resulting Javascript is also present.

Where possible, caterwaul's libraries make it simple to switch between semantic and notational abstraction. This is done by imposing scoped contexts on various subexpressions. The context determines how trees are interpreted. This homoiconicity is also present in Common Lisp, though this language is more closed under quotation than Lisp's `defmacro`. For example:

```
;;; Common Lisp: macro body is unquoted, so explicit quotation
;;; is used to change context
(defmacro foo (x)
  `(1+ ,x))
(foo 5)

# Caterwaul: both sides of the macro definition are equally
# quoted, so macro definitions are maps instead of flat-maps
foo 5 -wh [foo _x = _x + 1]
```

The main difference between these approaches is that caterwaul's output format is a statically-known subset of all supported operators, so the toplevel is implicitly quasiquoted. These heuristics are more difficult to implement in Lisp due to its more general monomorphic consing and mutable global symbol table.