**A Summary of “Sweeten Your JavaScript: Hygienic Macros for ES5”**

Macro systems are used extensively in programming languages such as Lisp and Scheme in large part due to both languages’ use of fully-delimited, symbolic (s-) expressions, which make source code easy to manipulate via macros. In contrast, languages with ambiguous grammars are less conducive to macros since the compiler’s lexers and parser are intertwined with messages passing back and forth between them. For example, in JavaScript, there is lexical ambiguity around the forward slash (“/”) symbol as it may represent a regular expression or a divide operation.

This paper introduces Sweet.js, which is a macro system for JavaScript whose primary contribution is a reader that correctly distinguish between division operations and regular expressions across the entire ECMAScript5 (ES5) specification; this reader sits between the lexer and parser to eliminate the need for the bidirectional communication between the lexer and parser. The reader outputs a sequence of *token trees* (similar to s-expressions)

While most macro systems are primarily used for prefix macros (where the macro identifier appears before the matching syntax), Sweet.js also supports *custom operators* (which were introduced by the Honu programming language). Sweet.js also introduces the use of *infix macros* to overcome some of the limitations of infix operators by enabling matching of syntax before and after the macro identifier.

In a fully delimited language like Scheme, a token tree is sufficient to implement an expressive macro system. Partially or undelimited languages like JavaScript require additional structure to enable the use of macros. Sweet.js leverages the technique of enforestation first introduced in the Honu programming language, whereby token trees are transformed into term trees by progressively recognizing and grouping syntax forms (e.g. literals, identifiers, expressions, statements, etc.). This technique in turn fully deliminates the syntax.

While this paper only explicitly describes how to implement the proposed scheme in JavaScript, the approach of using token tokens to store lexical history can be applied to other ambiguous grammar languages. For example, Perl shares ambiguity around the forward slash (“/”) while the Rust language has ambiguity when parsing the less than (“<”) symbol. These types of ambiguities that necessitate intertwining of the lexer and parser can be resolved using the techniques proposed by Disney *et. al.*.