A Summary of "Sweeten Your JavaScript: Hygienic Macros for ES5"

Macro systems are used extensively in fully delimited programming languages that are founded on the use of symbolic (s-) expressions, as these s-expressions make the input source code easy to manipulate; prototypical language examples include Lisp and Scheme. In contrast, languages with ambiguous grammars are less conducive to macros since the compiler's lexer and parser are intertwined, with communication 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.

Disney *et. al.*'s paper introduces Sweet.js, which is a macro system for JavaScript whose primary contribution is a reader scheme that correctly distinguishes between division operations and regular expressions across the entire ECMAScript5 (ES5) specification. This reader sits between the lexer and parser and eliminates the need for the bidirectional communication between those two components. The reader outputs a sequence of *token trees* (similar to s-expressions). 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 around the token tree 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 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 too were introduced in Honu). What is more, Sweet.js introduces the use of *infix macros* to overcome some of the limitations of infix operators by enabling the matching of syntax both before and after the macro identifier.

Within a macro itself, Sweet.js supports two primary styles, namely *rule* and *case* based syntax. Rule macros allow an input to be greedily matched to the large possible pattern among a set of expressions and are primarily used for term rewriting. In contrast, case macros allow for macro-based JavaScript code to create and manipulate the input. An example use of a case macro that is proposed in the paper is one that reads a file and stores the file contents into a string. Both rule and case macros support the use of *pattern variables* (denoted by a preceding dollar sign "$") which allow Sweet.js to bind input syntax to a variable for use within the macro's template section.

While Disney *et. al.*'s paper only explicitly describes how to implement the proposed scheme in JavaScript, the approach of using token trees to store lexical history can be applied to other languages with ambiguous grammars. Additional examples where this technique could be used include Perl's similar ambiguity around forward slash ("/") as well as Rust's ambiguity when parsing the less than ("<") symbol.