Sweeten Your JavaScript: Hygienic Macros for ES5

Tim Disney
UC Santa Cruz

Nathan Faubion
Flow Corp.

David Herman
Mozilla Corp.

Cormac Flanagan
UC Santa Cruz

Abstract

Lisp and Scheme have demonstrated the power of macros to enable programmers to evolve and craft languages. In languages with more complex syntax, macros have had less success. In part, this has been due to the difficulty in building expressive hygienic macro systems for such languages. JavaScript in particular presents unique challenges for macro systems due to ambiguities in the lexing stage that force the JavaScript lexer and parser to be intertwined.

In this paper we present a novel solution to the lexing ambiguity of JavaScript that enables us to cleanly separate the JavaScript lexer and parser by recording enough history during lexing to resolve ambiguities. We give an algorithm for this solution along with a proof that it does in fact correctly resolve ambiguities in the language. Though the algorithm and proof we present is specific to JavaScript, the general technique can be applied to other languages with ambiguous grammars. With lexer and parser separated, we then implement an expressive hygienic macro system for JavaScript called sweet.js.

1. Introduction

Expressive macro systems have a long history in the design of extensible programming languages going back to Lisp and Scheme [15, 22] as a powerful tool that enables programmers to craft their own languages.

While macro systems have found success in many Lisp-derived languages, they have not been widely adopted in languages such as JavaScript. In part, this failure is due to the difficulty of implementing macro systems for languages without fully delimited s-expressions. A key feature of a sufficiently expressive macro system is the ability for macros to manipulate unparsed and unexpanded subexpressions. In a language with parentheses like Scheme, manipulating unparsed subexpressions is simple:

```
(if (> denom 0)
    (/ x denom)
    (error "divide by zero"))
```

The Scheme *reader* converts the source string into nested s-expressions, which macros can easily manipulate. Since each subexpression of the `if` form is a fully delimited s-expression, it is easy to implement `if` as a macro.

Conceptually, the Scheme compiler lexes a source string into a stream of tokens which are then read into s-expressions before being macro expanded and parsed into an abstract syntax tree.

$$\textit{lexer} \xrightarrow{\textit{Token}^*} \textit{reader} \xrightarrow{\textit{Sexpr}} \textit{expander/parser} \xrightarrow{\textit{AST}}$$

As a first step to designing a Scheme-like macro system for JavaScript, it is necessary to introduce a read step into the compiler pipeline. However, the design of a correct reader for full JavaScript turns out to be surprisingly subtle, due to ambiguities in how regular expression literals (such as `/[0-9]*/`) and the divide operator (`/`) should be lexed. In traditional JavaScript compilers, the parser and lexer are intertwined. Rather than run the entire program through the lexer once to get a sequence of tokens, the parser calls out to the lexer from a given grammatical context with a flag to indicate if the lexer should accept a regular expression or a divide operator, and the input character stream is tokenized accordingly. So if the parser is in a context that accepts a regular expression, the characters "`/x/`" will be lexed into the single token `/x/` otherwise it will lex into the individual tokens `/`, `x`, and `/`.

$$\textit{lexer} \underset{\textit{Token}^*}{\overset{\textit{feedback}}{\rightleftarrows}} \textit{parser} \xrightarrow{\textit{AST}}$$

Figure 1: The sweet.js editor

It is necessary to separate the parser and lexer in order to implement a macro system for JavaScript. Our JavaScript macro system, sweet.js, includes a separate reader that converts a sequence of tokens into a sequence of token trees (a little analogous to s-expressions) without feedback from the parser.

$$\textit{lexer} \xrightarrow{\textit{Token}^*} \textit{reader} \xrightarrow{\textit{TokenTree}^*} \textit{expander/parser} \xrightarrow{\textit{AST}}$$

This enables us to finally separate the JavaScript lexer and parser and build a fully hygienic macro system for JavaScript. The reader records sufficient history information in the form of token trees in order to correctly decide whether to parse a sequence of tokens `/x/g` as a regular expression or as division operators (as in `4.0/x/g`). Surprisingly, this history information needs to be remembered from arbitrarily far back in the token stream.

While the algorithm for resolving ambiguities we present in this paper is specific to JavaScript, the technique of recording history in the reader with token trees can be applied to other languages with ambiguous grammars.

Once JavaScript source has been correctly read, there are still a number of challenges to building an expressive macro system. The lack of parentheses in particular make writing declarative macro definitions difficult. For example, the `if` statement in JavaScript allows undelimited then and else branches:

```
if (denom > 0)
  x / denom;
else
  throw "divide by zero";
```

It is necessary to know where the then and else branches end to correctly implement an `if` macro but this is complicated by the lack of delimiters.

The solution to this problem that we take is by progressively building a partial AST during macro expansion. Macros can then match against and manipulate this partial AST. For example, an `if` macro could indicate that the then and else branches must be single statements and then manipulate them appropriately.

This approach, called *enforestation*, was pioneered by Honu [29, 30], which we adapt here for JavaScript[1]. In addition, we make two extensions to the Honu technique that enable more expressive macros to be built. First, as described in Section 4.2 we add support for infix macros, which allow macros to match syntax both before and after the macro identifier. Secondly, we implement the `invoke` pattern

[1] The syntax of Honu is similar to JavaScript but does not support regular expression literals, which simplifies their reader.

Figure 2: AST for Simplified JavaScript

$$
\begin{array}{rcl}
e \in \mathit{AST} & ::= & x \mid \texttt{/}r\texttt{/} \mid \texttt{\{x: }e\texttt{\}} \mid \texttt{(}e\texttt{)} \mid e\texttt{.x} \mid e\texttt{(}e\texttt{)} \\
& \mid & e \texttt{ / } e \mid e \texttt{ + } e \mid e \texttt{ = } e \mid \texttt{\{}e\texttt{\}} \mid \texttt{x:}e \mid \texttt{if (}e\texttt{) } e \\
& \mid & \texttt{return} \mid \texttt{return } e \\
& \mid & \texttt{function } x \texttt{ (}x\texttt{) \{}e\texttt{\}} \mid e\ e
\end{array}
$$

class, described in Section 4.3, which allows macro authors to extend the patterns used to match syntax.

Sweet.js is implemented in JavaScript and takes source code written with sweet.js macros and produces the expanded source that can be run in any JavaScript environment. The project web page[2] includes an interactive browser-based editor that makes it simple to try out writing macros without requiring any installation. Figure 1 shows the editor in action; a macro implementing classes is being edited in the left pane and the right pane continually shows the expanded output. There is already an active community using sweet.js to, for example, implement significant features from the upcoming ES6 version of JavaScript [24] or implement pattern matching in JavaScript [12].

2. Reading JavaScript

Parsers give structure to unstructured source code. In parsers without macro systems this is usually accomplished by a lexer (which converts a character stream to a token stream) and a parser (which converts the token stream into an AST according to a context-free grammar). A system with macros must implement a macro *expander* that sits between the lexer and parser. Some macros systems, such as the C preprocessor [19], work over just the token stream. However, to implement truly expressive Scheme-like macros that can manipulate groups of unparsed tokens, it is necessary to structure the token stream via a *reader*, which performs delimiter matching and enables macros to manipulate delimiter-grouped tokens.

As mentioned in the introduction, the design of a correct reader for JavaScript is surprisingly subtle due to ambiguities between lexing regular expression literals and the divide operator. This disambiguation is critical to the correct implementation of read because delimiters can appear inside of a regular expression literal. If the reader failed to distinguish between a regular expression/divide operator, it could result in incorrectly matched delimiters.

```
function makeRegex() {
  // results in a parse error if the
  // first / is incorrectly read as divide
  return /}/;
}
```

A key novelty in sweet.js is the design and implementation of a reader that correctly distinguishes between regular expression literals and the divide operator for full ES5 JavaScript[3]. For clarity of presentation, this paper describes the implementation of read for the subset of JavaScript shown in Figure 4, which retains the essential complications of the full version of read.

2.1 Formalism

In our formalism in Figure 3, read takes a *Token* sequence. Tokens are the output of a very simple lexer, which we do not define here. This lexer does not receive feedback from the parser like the ES5 lexer does, and so does not distinguish between regular expressions and the divide operator. Rather it simply lexes slashes into the ambiguous `/` token. Tokens also include keywords, puncutators, the (unmatched) delimiters, and variable identifiers.

$$
\begin{array}{lcl}
\mathit{Punctuator} & ::= & \texttt{/} \mid \texttt{+} \mid \texttt{:} \mid \texttt{;} \mid \texttt{=} \mid \texttt{.} \\
\mathit{Keyword} & ::= & \texttt{return} \mid \texttt{function} \mid \texttt{if} \\
\mathit{Token} & ::= & x \mid \mathit{Punctuator} \mid \mathit{Keyword} \\
& \mid & \texttt{\{} \mid \texttt{\}} \mid \texttt{(} \mid \texttt{)} \\
x, y & \in & \mathit{Variable} \\
s & \in & \mathit{Token}^*
\end{array}
$$

The job of read is then to produce a correct *TokenTree* sequence. Token trees include regular expression literals $\texttt{/}r\texttt{/}$, where r is the regular expression body. We simplify regular expression bodies to just a variable and the individual delimiters, which captures the essential problems of parsing regular expressions. Token trees also include fully matched delimiters with nested token tree sequences $\underline{\texttt{(}t\texttt{)}}$ and $\underline{\texttt{\{}t\texttt{\}}}$ rather than individual delimiters (we write token tree delimiters with an underline to distinguish them from token delimiters).

$$
\begin{array}{lcl}
k \in \mathit{TokenTree} & ::= & x \mid \mathit{Punctuator} \mid \mathit{Keyword} \\
& \mid & \texttt{/}r\texttt{/} \mid \underline{\texttt{(}t\texttt{)}} \mid \underline{\texttt{\{}t\texttt{\}}} \\
r \in \mathit{RegexPat} & ::= & x \mid \texttt{\{} \mid \texttt{\}} \mid \texttt{(} \mid \texttt{)} \\
t, p & \in & \mathit{TokenTree}^*
\end{array}
$$

Each token and token tree also carries their line number from the original source string. Line numbers are needed because there are edge cases in the JavaScript grammar where newlines influence parsing. For example, the following function returns the object literal `{x: y}` as expected.

```
function f(y) {
  return { x: y }
}
```

However, adding a newline causes this function to return `undefined`, because the grammar calls for an implicit semicolon to be inserted after the `return` keyword.

```
function g(y) {
  return
  { x: y }
}
```

[2] `http://sweetjs.org`

[3] Our implementation also has initial support for code written in the upcoming ES6 version of JavaScript.

Figure 3: Simplified Read Algorithm

$$
\begin{array}{lcl}
\textit{Punctuator} & ::= & \texttt{/} \mid \texttt{+} \mid \texttt{:} \mid \texttt{;} \mid \texttt{=} \mid \texttt{.} \\
\textit{Keyword} & ::= & \texttt{return} \mid \texttt{function} \mid \texttt{if} \\
\textit{Token} & ::= & x \mid \textit{Punctuator} \mid \textit{Keyword} \\
 & \mid & \texttt{\{} \mid \texttt{\}} \mid \texttt{(} \mid \texttt{)} \\
k \in \textit{TokenTree} & ::= & x \mid \textit{Punctuator} \mid \textit{Keyword} \\
 & \mid & \texttt{/}r\texttt{/} \mid \underline{(t)} \mid \underline{\{t\}} \\
r \in \textit{RegexPat} & ::= & x \mid \texttt{\{} \mid \texttt{\}} \mid \texttt{(} \mid \texttt{)} \\
x & \in & \textit{Variable} \\
s & \in & \textit{Token}^* \\
t, p & \in & \textit{TokenTree}^*
\end{array}
$$

$$
\begin{array}{l}
\text{isExprPrefix} : \textit{TokenTree}^* \to \textit{Bool} \to \textit{Int} \to \textit{Bool} \\
\text{isExprPrefix}(\epsilon,\ \textit{true},\ l) = \textit{true} \\
\text{isExprPrefix}(p \cdot \texttt{/},\ b,\ l) = \textit{true} \\
\text{isExprPrefix}(p \cdot \texttt{+},\ b,\ l) = \textit{true} \\
\text{isExprPrefix}(p \cdot \texttt{=},\ b,\ l) = \textit{true} \\
\text{isExprPrefix}(p \cdot \texttt{:},\ b,\ l) = b \\
\text{isExprPrefix}(p \cdot \texttt{return}^l,\ b,\ l') = \textit{false} \quad \textit{if } l \neq l' \\
\text{isExprPrefix}(p \cdot \texttt{return}^l,\ b,\ l') = \textit{true} \quad \textit{if } l = l' \\
\text{isExprPrefix}(p,\ b,\ l) = \textit{false}
\end{array}
$$

$$
\begin{array}{l}
\text{read} : \textit{Token}^* \to \textit{TokenTree}^* \to \textit{Bool} \to \textit{TokenTree}^* \\
\text{read}(\texttt{/} \cdot r \cdot \texttt{/} \cdot s,\ \epsilon,\ b) = \texttt{/}r\texttt{/} \cdot \text{read}(s,\ \texttt{/}r\texttt{/},\ b) \\
\text{read}(\texttt{/} \cdot r \cdot \texttt{/} \cdot s,\ p \cdot k,\ b) = \texttt{/}r\texttt{/} \cdot \text{read}(s,\ p \cdot k \cdot \texttt{/}r\texttt{/},\ b) \\
\quad \textit{if } k \in \textit{Punctuator} \cup \textit{Keyword} \\
\text{read}(\texttt{/} \cdot r \cdot \texttt{/} \cdot s,\ p \cdot \texttt{if} \cdot \underline{(t)},\ b) = \texttt{/}r\texttt{/} \cdot \text{read}(s,\ p \cdot \texttt{if} \cdot \underline{(t)} \cdot \texttt{/}r\texttt{/},\ b) \\
\text{read}(\texttt{/} \cdot r \cdot \texttt{/} \cdot s,\ p \cdot \texttt{function}^l \cdot x \cdot \underline{(t)} \cdot \underline{\{t'\}},\ b) = \texttt{/}r\texttt{/} \cdot \text{read}(s,\ p \cdot \texttt{function}^l \cdot x \cdot \underline{(t)} \cdot \underline{\{t'\}} \cdot \texttt{/}r\texttt{/},\ b) \\
\quad \textit{if } \text{isExprPrefix}(p,\ b,\ l) = \textit{false} \\
\text{read}(\texttt{/} \cdot r \cdot \texttt{/} \cdot s,\ p \cdot \underline{\{t\}}^l,\ b) = \texttt{/}r\texttt{/} \cdot \text{read}(s,\ p \cdot \underline{\{t\}}^l \cdot \texttt{/}r\texttt{/},\ b) \\
\quad \textit{if } \text{isExprPrefix}(p,\ b,\ l) = \textit{false} \\
\\
\text{read}(\texttt{/} \cdot s,\ p \cdot x,\ b) = \texttt{/} \cdot \text{read}(s,\ p \cdot x \cdot \texttt{/},\ b) \\
\text{read}(\texttt{/} \cdot s,\ p \cdot \texttt{/}x\texttt{/},\ b) = \texttt{/} \cdot \text{read}(s,\ p \cdot \texttt{/}x\texttt{/} \cdot \texttt{/},\ b) \\
\text{read}(\texttt{/} \cdot s,\ p \cdot \underline{(t)},\ b) = \texttt{/} \cdot \text{read}(s,\ p \cdot \underline{(t)} \cdot \texttt{/},\ b) \\
\text{read}(\texttt{/} \cdot s,\ p \cdot \texttt{function}^l \cdot x \cdot \underline{(t)} \cdot \underline{\{t'\}},\ b) = \texttt{/} \cdot \text{read}(s,\ p \cdot \texttt{function}^l \cdot x \cdot \underline{(t)} \cdot \underline{\{t'\}} \cdot \texttt{/},\ b) \\
\quad \textit{if } \text{isExprPrefix}(p,\ b,\ l) = \textit{true} \\
\text{read}(\texttt{/} \cdot s,\ p \cdot \underline{\{t\}}^l,\ b) = \texttt{/} \cdot \text{read}(s,\ p \cdot \underline{\{t\}}^l \cdot \texttt{/},\ b) \\
\quad \textit{if } \text{isExprPrefix}(p,\ b,\ l) = \textit{true} \\
\\
\text{read}(\texttt{(} \cdot s \cdot \texttt{)} \cdot s',\ p,\ b) = \underline{(t)} \cdot \text{read}(s',\ p \cdot \underline{(t)},\ b) \\
\quad \textit{where } s \textit{ contains no unmatched } \texttt{)} \qquad \textit{where } t = \text{read}(s,\ \epsilon,\ \textit{false}) \\
\text{read}(\texttt{\{}^l \cdot s \cdot \texttt{\}} \cdot s',\ p,\ b) = \underline{\{t\}}^l \cdot \text{read}(s',\ p \cdot \underline{\{t\}}^l,\ b) \\
\quad \textit{where } s \textit{ contains no unmatched } \texttt{\}} \qquad \textit{where } t = \text{read}(s,\ \epsilon,\ \text{isExprPrefix}(p,\ b,\ l)) \\
\text{read}(x \cdot s,\ p,\ b) = x \cdot \text{read}(s,\ p \cdot x,\ b) \\
\text{read}(\epsilon,\ p,\ b) = \epsilon
\end{array}
$$

For clarity of presentation, we leave token line numbers implicit unless we require them, in which case we use the notation $\texttt{\{}^l$ where l is a line number.

We write a token sequence by separating elements with a dot so the source string "`foo(/)/)`" is lexed into a sequence of six tokens $\texttt{foo} \cdot \texttt{(} \cdot \texttt{/} \cdot \texttt{)} \cdot \texttt{/} \cdot \texttt{)}$. The equivalent token tree sequence is $\texttt{foo} \cdot \underline{\texttt{(} \cdot \texttt{/)/} \cdot \texttt{)}}$.

2.2 Read Algorithm

The key idea of read is to maintain a prefix of already read token trees. When the reader comes to a slash and needs to decide if it should read the slash as a divide token or the start of a regular expression literal, it consults the prefix. Looking back at most five tokens trees in the prefix is sufficient to disambiguate the slash token. Note that this may correspond to looking back an unbounded distance in the original token stream.

Some of the cases of read are relatively obvious. For example, if the token just read was one of the binary operators (e.g. the + in $\texttt{f} \cdot \texttt{+} \cdot \texttt{/} \cdot \texttt{\}} \cdot \texttt{/}$) the slash will always be the start of a regular expression literal.

Other cases require additional context to disambiguate. For example, if the previous token tree was a parentheses (e.g. $\texttt{foo} \cdot \texttt{(} \cdot \texttt{x} \cdot \texttt{)} \cdot \texttt{/} \cdot \texttt{y}$) then slash will be the divide operator, *unless* the token tree before the parentheses was the keyword

Figure 4: Simplified ES5 Grammar

$$
\begin{array}{lcl}
\mathit{PrimaryExpr}_{x} & ::= & x \\
\mathit{PrimaryExpr}_{/r/} & ::= & \texttt{/} \cdot x \cdot \texttt{/} \\
\mathit{PrimaryExpr}_{\texttt{\{x:}e\texttt{\}}} & ::= & \texttt{\{} \cdot \texttt{x} \cdot \texttt{:} \cdot \mathit{AssignExpr}_{e} \cdot \texttt{\}} \\
\mathit{PrimaryExpr}_{\texttt{(}e\texttt{)}} & ::= & \texttt{(} \cdot \mathit{AssignExpr}_{e} \cdot \texttt{)} \\
\\
\mathit{MemberExpr}_{e} & ::= & \mathit{PrimaryExpr}_{e} \\
\mathit{MemberExpr}_{e} & ::= & \mathit{Function}_{e} \\
\mathit{MemberExpr}_{e\texttt{.x}} & ::= & \mathit{MemberExpr}_{e} \cdot \texttt{.} \cdot \texttt{x} \\
\\
\mathit{CallExpr}_{e\ \texttt{(}e'\texttt{)}} & ::= & \mathit{MemberExpr}_{e} \cdot \texttt{(} \cdot \mathit{AssignExpr}_{e'} \cdot \texttt{)} \\
\mathit{CallExpr}_{e\ \texttt{(}e'\texttt{)}} & ::= & \mathit{CallExpr}_{e} \cdot \texttt{(} \cdot \mathit{AssignExpr}_{e'} \cdot \texttt{)} \\
\mathit{CallExpr}_{e\texttt{.x}} & ::= & \mathit{CallExpr}_{e}\ \texttt{.}\ \texttt{x} \\
\\
\mathit{BinaryExpr}_{e} & ::= & \mathit{CallExpr}_{e} \\
\mathit{BinaryExpr}_{e\ \texttt{/}\ e'} & ::= & \mathit{BinaryExpr}_{e} \cdot \texttt{/} \cdot \mathit{BinaryExpr}_{e'} \\
\mathit{BinaryExpr}_{e\ \texttt{+}\ e'} & ::= & \mathit{BinaryExpr}_{e} \cdot \texttt{+} \cdot \mathit{BinaryExpr}_{e'} \\
\\
\mathit{AssignExpr}_{e} & ::= & \mathit{BinaryExpr}_{e} \\
\mathit{AssignExpr}_{e\ \texttt{=}\ e'} & ::= & \mathit{CallExpr}_{e} \cdot \texttt{=} \cdot \mathit{AssignExpr}_{e'} \\
\\
\mathit{StmtList}_{e} & ::= & \mathit{Stmt}_{e} \\
\mathit{StmtList}_{e\ e'} & ::= & \mathit{StmtList}_{e} \cdot \mathit{Stmt}_{e'} \\
\\
\mathit{Stmt}_{\texttt{\{}e\texttt{\}}} & ::= & \texttt{\{} \cdot \mathit{StmtList}_{e} \cdot \texttt{\}} \\
\mathit{Stmt}_{\texttt{x:}\ e} & ::= & \texttt{x} \cdot \texttt{:} \cdot \mathit{Stmt}_{e} \\
\mathit{Stmt}_{e} & ::= & \mathit{AssignExpr}_{e} \cdot \texttt{;} \qquad \textit{where lookahead} \neq \texttt{\{}\ \textit{or}\ \texttt{function} \\
\mathit{Stmt}_{\texttt{if (}e\texttt{)}\ e'} & ::= & \texttt{if} \cdot \texttt{(} \cdot \mathit{AssignExpr}_{e} \cdot \texttt{)} \cdot \mathit{Stmt}_{e'} \\
\mathit{Stmt}_{\texttt{return}} & ::= & \texttt{return} \\
\mathit{Stmt}_{\texttt{return}\ e} & ::= & \texttt{return} \cdot \text{[no line terminator here]}\ \mathit{AssignExpr}_{e} \cdot \texttt{;} \\
\\
\mathit{Function}_{\texttt{function}\ x\ \texttt{(}x'\texttt{)\ \{}e\texttt{\}}} & ::= & \texttt{function} \cdot x \cdot \texttt{(} \cdot x' \cdot \texttt{)} \cdot \texttt{\{} \cdot \mathit{SourceElements}_{e} \cdot \texttt{\}} \\
\\
\mathit{SourceElement}_{e} & ::= & \mathit{Stmt}_{e} \\
\mathit{SourceElement}_{e} & ::= & \mathit{Function}_{e} \\
\\
\mathit{SourceElements}_{e} & ::= & \mathit{SourceElement}_{e} \\
\mathit{SourceElements}_{e\ e'} & ::= & \mathit{SourceElements}_{e} \cdot \mathit{SourceElement}_{e'} \\
\\
\mathit{Program}_{e} & ::= & \mathit{SourceElements}_{e} \\
\mathit{Program} & ::= & \epsilon
\end{array}
$$

`if`, in which case it is actually the start of a regular expression (since single statement `if` bodies do not require braces).

```
if (x) /}/   // regex
```

One of the most complicated cases is a slash after curly braces. Part of the complication here is that curly braces can either indicate an object literal (in which case the slash should be a divide) or a block (in which case the slash should be a regular expression), but even more problematic is that both object literals and blocks with labeled statements can nest. For example, in the following code snippet the outer curly brace is a block with a labeled statement `x`, which is another block with a labeled statement `y` followed by a regular expression literal.

```
{
  x:{y: z} /}/  // regex
}
```

But if we change the code slightly, the outer curly braces become an object literal and `x` is a property so the inner curly braces are also an object literal and thus the slash is a divide operator.

```
o = {
  x:{y: z} /x/g  // divide
}
```

While it is unlikely that a programmer would attempt to intentionally perform division on an object literal, it is not a parse error. In fact, this is not even a runtime error since JavaScript will implicitly convert the object to a number (technically `NaN`) and then perform the division (yielding `NaN`).

The reader handles these cases by checking if the prefix of a curly brace forces the curly to be an object literal or a statement block and then setting a boolean flag to be used while reading the tokens inside of the braces.

Based on this discussion, our reader is implemented as a function that takes a sequence of tokens, a prefix of previously read token trees, a boolean indicating if the token stream currently being read is inside an object literal, and returns a sequence of token trees.

$$\text{read} : \textit{Token}^* \to \textit{TokenTree}^* \to \textit{Bool} \to \textit{TokenTree}^*$$

The implementation of read shown in Figure 3 includes an auxiliary function isExprPrefix used to determine if the prefix for a curly brace indicates that the braces should be part of an expression (i.e. the braces are an object literal) or if they should be a block statement.

Interestingly, the isExprPrefix function must also be used when the prefix before a slash contains a function definition. This is because there are two kinds of function definitions in JavaScript, function expressions and function declarations, and these also affect how slash is read. For example, a slash following a function declaration is always the start of a regular expression:

```
function f() {}
/}/   // regex
```

However, a slash following a function expression is a divide operator:

```
x = function f() { }
/y/g   // divide
```

As in the object literal case, it is unlikely that a programmer would attempt to intentionally divide a function expression but it is not an error to do so.

2.3 Proving Read

To show that our read algorithm correctly distinguishes divide operations from regular expression literals, we show that a parser defined over normal tokens produces the same AST as a parser defined over the token trees produced from read.

The parser for normal tokens is defined in Figure 4, and generates ASTs in the abstract syntax shown in Figure 2. A parser for the nonterminal *Program* is a function from a sequence of tokens to an AST.

$$\textit{Program} :: \textit{Token}^* \to \textit{AST}$$

We use notation whereby the grammar production $\textit{Program}_e ::= \textit{SourceElements}_e$ means to match the input sequence with $\textit{SourceElements}_e$ and produce the resulting AST e.

Note that the grammar we present here is a simplified version of the grammar specified in the ECMAScript 5.1 standard [21] and many nonterminal names are shortened versions of nonterminals in the standard. It is mostly straightforward to extend the algorithm presented here to the full sweet.js implementation for ES5 JavaScript.

In addition to the *Program* parser just described, we also define a parser *Program'* that works over token trees. The rules of the two parsers are identical, except that all rules with delimiters and regular expression literals change as follows:

$$\begin{array}{lcl} \textit{PrimaryExpr}_{/r/} & ::= & / \cdot r \cdot / \\ \textit{PrimaryExpr'}_{/r/} & ::= & /r/ \\ \textit{PrimaryExpr}_{(e)} & ::= & (\cdot \textit{AssignExpr}_e \cdot) \\ \textit{PrimaryExpr'}_{(e)} & ::= & \underline{(\textit{AssignExpr'}_e)} \end{array}$$

To prove that read is correct, we show that the following two parsing strategies give identical behavior:

- The traditional parsing strategy takes a token sequence s and parses s into an AST e using the traditional parser $\textit{Program}_e$.
- The second parsing strategy first reads s into a token tree sequence $\text{read}(s,\ \epsilon,\ \textit{false})$, and then parses this token tree sequence into an AST e via $\textit{Program'}_e$.

Theorem 1 (Parse Equivalence)**.**
$\forall s \in \textit{Token}^*.$
$s \in \textit{Program}_e \Leftrightarrow \textit{read}(s,\ \epsilon,\ \textit{false}) \in \textit{Program'}_e$

Proof. The proof proceeds by induction on ASTs to show that parse equivalence holds between all corresponding nonterminals in the two grammars. We present the details of this proof in an appendix available online [10]. □

3. Writing Macros

The sweet.js system provides two kinds of macros: *rule* macros (analogous to `syntax-rules` in Scheme [7]) and *case* macros (analogous to `syntax-case` in Scheme [20]). Rule macros are the simpler of the two and work by matching on a pattern and generating a template:

```
macro <name> {
  rule { <pattern> } => { <template> }
}
```

For example, the following macro introduces a new function definition form:

```
macro def {
  rule {
    $name ($params (,) ...) { $body ... }
  } => {
    function $name ($params ...) {
      $body ...
    }
  }
}
def id (x) { return x; }
// expands to:
// function id (x) { return x; }
```

The pattern is matched against the syntax following the macro name. Identifiers in a pattern that begin with `$` are *pattern variables* and bind the syntax they match in the template (identifiers that do not begin with `$` are matched literally). The ellipses (`$params (,)...`) mean match zero or more tokens separated by commas.

The above example shows the power of matching delimited groups of syntax (i.e. matching all the tokens inside the function body). In order for macros to be convenient in a language like JavaScript, it is necessary to have the ability to match logical groupings of syntax that are not fully delimited. For example, consider the `let` macro:

```
macro let {
  rule { $id = $init:expr } => {
    var $id = $init
  }
}
let x = 40 + 2;
// expands to:
// var x = 40 + 2;
```

The initialization of a `let` can be an arbitrary expression so we use the *pattern class* `:expr` to greedily match the largest possible expression, so in this example the entire expression `40 + 2` is bound to `$init`.

Note that sweet.js does not provide a non-greedy form of `:expr` so there is no way, for example, to match the pattern `$x:expr + 10`. In our experience to date it appears rare to need to match on syntax that extends an expression and so have avoided increasing the complexity of the implementation and pattern matching language by adding a non-greedy form.

Along with the template-based rule macros, sweet.js also provides the more powerful procedural *case* macros. While rule macros just provide term rewriting, case macros allow macro authors to use JavaScript code to procedurally create and manipulate syntax. For example, the following macro creates a string from the contents of a file.

```
macro fromFile {
  case {_ ($path) } => {
    var fname = unwrapSyntax(#{$path});
    var f = readFile(fname);
    letstx $content = [makeValue(f, #{here})];
    return #{$content}
  }
}
var s = fromFile ("./file.txt")
// expands to:
// s = "contents of file";
```

Syntax bound in the macro pattern can be referenced inside of JavaScript code using the notation `#{ ... }`. Here we take the file name token matched by the macro (`#{$path}`), unwrap it (like in Scheme, tokens that can be manipulated by a case macro are represented as a *syntax object* which wraps a token with its lexical context used to maintain hygiene so unwrapping extracts just the token), and read[4] the file into a variable. The `makeValue` function is used to create a string syntax object (`#{here}` is just a placeholder for the lexical context used for hygiene) from the context of the file and `letstx` (analogous to `with-syntax` in Scheme) binds that new syntax object to a pattern variable `$content` that can be used inside of a template.

4. Enforestation

The token tree structure produced by the reader is sufficient to implement an expressive macro system for a fully delimited language like Scheme. However since most of the syntax forms in a language like JavaScript are only partially delimited, it is necessary to provide additional structure during expansion that allows macros to manipulate undelimited or partially delimited groups of tokens. As an example, consider the `let` macro described earlier:

```
macro let {
  rule { $id = $init:expr } => {
    var $id = $init
  }
}
let x = 40 + 2;
// expands to:
// var x = 40 + 2;
```

Like many syntactic forms in JavaScript, the variable initialization expression is an undelimited group of tokens. Building an expressive macro system requires that the macro can match and manipulate patterns such as an expression.

Sweet.js groups tokens by transforming a token tree into a *term tree* through a technique pioneered by the Honu language called *enforestation* [29]. Enforestation works by progressively recognizing and grouping (potentially undelimited)

[4] Note that `readFile` will depend on the particular JavaScript environment in which sweet.js is being run (e.g. in node.js it might be `fs.readFile` and in the browser it might be an `XMLHttpRequest`).

syntax forms (e.g. literals, identifiers, expressions, and statements) during expansion. Essentially, enforestation delimits undelimited syntax.

A term tree is a kind of proto-AST that represents a partial parse of the program. As the expander passes through the token trees, it creates term trees that contain unexpanded sub trees that will be expanded once all macro definitions have been discovered in the current scope (as discussed in Section 5.1).

For an example of how enforestation progresses, consider the following use of the `let` macro:

```
macro let {
  rule { $id = $init:expr } => {
    var $id = $init
  }
}
function foo(x) {
  let y = 40 + 2;
  return x + y;
}
foo(100);
```

Enforestation begins by loading the `let` macro into the macro environment and converting the function declaration into a term tree (we use angle brackets to denote the term tree data structure). Notice that the body of the function has not yet been enforested in the first pass.

```
<fn: foo,
 args: (x),
 body: {
   let y = 40 + 2;
   return x + y;
 }>
foo(100);
```

Next, a term tree is created for the function call.

```
<fn: foo,
 params: (x),
 body: {
   let y = 40 + 2;
   return x + y;
 }>
<call: foo, args: (100)>
```

On the second pass through the top level scope the expander moves into the function body. The use of `let` is expanded away and the `var` and `return` term trees are created.

```
<fn: foo,
 params: (x),
 body: {
   <var: x, init: <op: +, left: 40, right: 2>
   <return: <op: +, left: x right: y>
 }>
<call: foo, args: (100)>
```

The additional structure provided by the term trees allow macros to match undelimited groups of tokens like binary expressions, such as `<op: +, left: 40, right: 2>`.

4.1 Custom Operators

The macros we have described so far are prefix macros: the macro identifier appears before the syntax that it matches. While prefix macros are sufficient for many syntax forms, other forms require additional flexibility.

Following Honu, sweet.js provides *custom operators*, which are user definable unary and binary operators. Custom operators provide the ability to set custom precedence and associativity along with a syntax transformation. For example, using sweet.js the exponentiation operator can be defined as:

```
operator (^^) 14 right
{ $base, $exp } => #{ Math.pow($base, $exp) }

y + x ^^ 10 ^^ 100 - z
// expands to:
// y + Math.pow(x, Math.pow(10, 100)) - z;
```

4.2 Infix Macros

While powerful, custom operators are limited in that their operands must be expressions, they cannot match on arbitrary syntax. This restriction means that with just prefix macros and custom operators it is impossible to define syntax forms like the arrow functions in the upcoming ES6 version of JavaScript:

```
id = (x) => x
// equivalent to:
// id = (function(x) { return x; });
```

One of the primary goals of sweet.js is to enable the kinds of syntax extension previously only done by the standardization committee. But, prefix macros are not capable of implementing syntax extensions like arrow functions since the macro name (`=>`) is in the middle of its syntax arguments and the syntax on the left hand side of the arrow is not an expression (meaning custom operators are not sufficient to implement arrow functions).

To address this limitation, sweet.js introduces *infix macros*, which allow a macro to match syntax that comes before and after the macro identifier. Infix macros are defined by adding the keyword `infix` after the `rule` or `case` keyword and placing a pipe (`|`) in the pattern where the macro name would appear (the pipe can be thought of as a cursor into the token stream).

For example, the following infix macro implements arrow functions[5]:

```
macro => {
  rule infix {
    ($params ...) | { $body:expr }
  } => {
    function ($params ...) {
      return $body;
    }
  }
}

var id = (x) => x
// expands to:
// var id = function (x) { return x; }
```

[5] For simplicity, this example does not bind `this` in the same manner as full ES6 arrow functions, though it is straightforward to do so.

Standard prefix macro transformers are invoked with the sequence of tokens following the macro identifier and then return a modified sequence of tokens that are then expanded:

$$transformer : TokenTree^* \rightarrow TokenTree^*$$

To implement infix macros, we modify the type of a transformer to take two arguments, one for the tokens that precede the macro identifier and the other with the tokens that follow. The transformer may then consume from either end as needed, yielding new preceding and following tokens:

$$\begin{aligned} transfomer_{infix} \quad &: \quad (TokenTree^*, TokenTree^*) \\ &\rightarrow \quad (TokenTree^*, TokenTree^*) \end{aligned}$$

A naive implementation of this transformer type will lead to brittle edge cases. For example:

```
bar(x) => x
```

Here the `=>` macro is juxtaposed next to a function call, which we did not intend to be valid syntax. The naive expansion results in unparsable code:

```
bar function(x) { return x; }
```

In more subtle cases, a naive expansion might result in code that actually parses but has incorrect semantics, leading to a debugging nightmare.

To avoid this problem we provide the macro transformer with both the previous tokens and their term tree representation.

$$\begin{aligned} transfomer_{infix} \quad &: \quad ((TokenTree^*, TermTree^*), TokenTree^*) \\ &\rightarrow \quad (TokenTree^*, TokenTree^*) \end{aligned}$$

We verify that an infix macro only matches previous tokens within boundaries delimited by the term trees. In our running example:

```
bar(x) => x
```

is first enforested to:

```
<call: bar, args: (x)> => x
```

before invoking the `=>` transformer with both the tokens `bar(x )` and the term tree `<call: bar, args: (x)>`. When the macro attempts to match just `(x)`, it detects that the parentheses splits the term tree `<call: bar, args: (x)>` and fails the match.

While infix macros fill the gap left by custom operators and allow us to write previously undefinable macros such as arrow functions, they also come with two limitations. First, there is no way to set custom precedence or associativity as with operators. It is unclear if this is a fundamental limitation or if there is some technique that might allow custom precedence and associativity for infix macros. The second limitation is that the preceding tokens to an infix macro must be first expanded. This means that any macros that occur before an infix macro will be invoked first.

This behavior introduces an asymmetry between the kinds of syntax an infix macro can match before its identifier and the kinds of syntax it can match after since the syntax following the identifier can contain unexpanded macros.

Even with these limitations infix, macros work as a powerful complement to custom operators and greatly extend the kinds of syntax forms that can be implemented with macros.

4.3 Invoke and Pattern Classes

Extensibility is the guiding design principle of any expressive macro system. Since the entire point of macros is to extend the expressive power of the language, so too should the macro system itself be extensible by users. To that end sweet.js provides a mechanism to extend the patterns used by macros to match their arguments.

As motivation, consider the following macro that matches against color options:

```
macro colors_options {
  rule { (red)   } => { ["#FF0000"] }
  rule { (green) } => { ["#00FF00"] }
  rule { (blue)  } => { ["#0000FF"] }
}
r = colors_options (red)
g = colors_options (green)
// expands to:
// r = ["#FF0000"]
// g = ["#00FF00"]
```

Macros can have multiple rules and the first rule to match (from top to bottom) is used. While this macro seems to work, attempting to generalize it quickly leads to a mess:

```
macro colors_options {
  rule { (red, red) } => {
    ["#FF0000", "#FF0000"]
  }
  rule { (red, green) } => {
    ["#FF0000", "#00FF00"]
  }
  rule { (red, blue) } => {
    ["#FF0000", "#0000FF"]
  }
  // ... etc.
}
r = colors_options (red, green)
g = colors_options (green, blue)
// expands to:
// r = ["#FF0000", "#00FF00"]
// g = ["#00FF00", "#0000FF"]
```

While it is possible to solve this problem through the use of case macros, the declarative intent is quickly lost in a mess of token manipulation code.

Our solution to this problem is the `:invoke` pattern class. This pattern class takes as a parameter a macro name which is inserted into the token tree stream before matching. If the inserted macro successfully matches its arguments, the result of its expansion is bound to the pattern variable. This makes declarative options simple to write:

```
macro color {
  rule { red   } => { "#FF0000" }
  rule { green } => { "#00FF00" }
  rule { blue  } => { "#0000FF" }
}
```

```
macro colors_options {
  rule { ($opt:invoke(color) (,) ...) } => {
    [$opt (,) ...]
  }
}
colors_options (red, green, blue, blue)
// expands to:
// ["#FF0000", "#00FF00", "#0000FF", "#0000FF"]
```

Here the `color` macro plays the role of enumerating the valid colors that can be matched and bound to `$opt`. If the token is not in one of the rules for `color` (e.g. `orange`), the `colors_options` macro will fail to match.

As a sweet added bit of sugar, the use of `:invoke` can be inferred by just using the name of a macro in scope (i.e. `$opt:color` is equivalent to `$opt:invoke(color)`).

```
macro color {
  rule { red   } => { "#FF0000" }
  rule { green } => { "#00FF00" }
  rule { blue  } => { "#0000FF" }
}
macro colors_options {
  rule { ($opt:color (,) ...) } => {
    [$opt (,) ...]
  }
}
colors_options (red, green, blue, blue)
// expands to:
// ["#FF0000", "#00FF00", "#0000FF", "#0000FF"]
```

By using `:invoke`, macro writers can encode patterns like alternates, optional tokens, keyword classes, numeric classes, and more in a declarative style.

5. Hygiene

Maintaining hygiene during macro expansion is perhaps the single most critical feature of an expressive macro system. The hygiene condition enables macros to be true syntactic *abstractions* by removing the burden of reasoning about a macro's implementation details from the user of a macro.

Our implementation of hygiene for sweet.js follows the Scheme approach [14, 20] of tracking the lexical context in each syntax object.

5.1 Macro Binding Limitation

Unfortunately, the syntax of JavaScript does place a limitations on our system that is not present in Scheme. The hygiene algorithm in Scheme allows definitions and uses of macros to be freely mixed in a given lexical scope. In particular, a macro can be used in an internal definition before its definition:

```
(define (foo)
  (define y (id 100))
  (define-syntax-rule (id x) x)
  (+ y y))
;; expands to:
;; (define (foo)
;;   (define y 100)
;;   (+ y y))
```

In order to support mixing use and definition, the Scheme expander must make multiple passes through a scope. The first pass registers any macro definitions and the second pass expands any uses of macros discovered during the first pass. Critically, during the first pass the expander does not expand inside internal definitions.

For example, after the first pass of expansion the above example will become:

```
(define (foo)
  (define y (id 100))
  (+ y y))
```

where the `id` macro has been registered in the macro environment. During the second pass, the expander will expand macros inside of internal definitions and so the example becomes:

```
(define (foo)
  (define y 100)
  (+ y y))
```

Scheme is able to defer expansion of macros inside of internal definitions because internal definitions are fully delimited. The expander can skip over all of the syntax inside of the delimiter because there actually is an *inside* to skip over. However, this is not true for JavaScript since `var` statements (JavaScript's equivalent of Scheme's internal definitions) are not delimited and so it is not possible to use a macro that has not yet been defined in a `var` statement in sweet.js.

For example, this will fail:

```
function foo() {
  var y = id 100;
  macro id { rule { $x } => { $x } }
  return y + y;
}
```

When the expander reaches the `var` statement during the first pass, it has not yet loaded the `id` macro into the macro environment and so `id 100` will be left unexpanded. Since `id 100` is not a valid expression, this will fail to parse. Without delimiters there is no way to defer expansion of the right-hand side of a `var` statement.

Note that at first glance it might appear that the semicolon could serve to delimit a var statement but this will not work for two reasons. First, a semicolon is just a token which might be comsumed by a macro. Second, the JavaScript specification calls for missing semicolons to be automatically inserted by the parser, which means it is not guaranteed that a semicolon will close every `var` statement.

Though the expander cannot defer expansion of `var` statements, it still does two passes so that the second pass can expand inside of delimiters. For example, a macro can be used inside of a function body that appears before the macro definition:

```
function foo() {
  return id 100;
}
macro id { rule { $x } => { $x } }
// expands to:
// function foo() {
//   return 100;
// }
```

While it is unfortunate that the syntax of JavaScript prevents fully general mixing of macro use and definition, the primary need for flexible macro definition locations is when writing mutually recursive macros, which is fully supported with our approach.

6. Discussion

The primary contribution this paper has presented is the separation of the lexer and the parser for JavaScript by way of a reader. Though the algorithm and proof presented here is JavaScript specific, the general technique can be leveraged in other languages in which ambiguities are present in the lexing phase.

For example, Perl 5 shares the `/` ambiguity with JavaScript [27] and in Rust [2] the token `<` is ambiguous—it can signify either the less than operator or the start of a template—and this ambiguity is resolved by intertwining the lexer and parser.

```
fn foo<T>(x: int, v: Vec<T>) -> bool {
    return x < 10;
}
```

The key insight of our approach is that the structure inherent in a token tree (in which delimited tokens are grouped together) provides a way to efficiently resolve ambiguities by looking at a relatively small prefix of token trees. The specific prefix required to disambiguate tokens such as `/` or `<` will depend on the grammar of the language in question but we hypothesize that our basic technique could be applied for other languages. Verifying this hypothesis is an interesting topic for future work.

In addition to enabling the construction of a macro system (the motivation for our work here), a correct reader also enables other tools that benefit from the additional structure provided by token trees such as robust syntax highlighters or editors that can correctly manipulate the structure of code while avoiding the expense of a full parse or expansion. The reader algorithm also allows traditional JavaScript parsers to be built with a cleaner design where the lexing and parsing stages are separated. The JavaScript parser Esprima [1] has incorporated our reader algorithm into their lexing stages to accomplish this separation.

7. Related Work

Macros have been extensively used and studied in Lisp [15, 28] and related languages for many years. Scheme in particular has embraced macros, pioneering the development of declarative definitions [22] and working out the hygiene conditions for term rewriting macros (rule macros) [7] and procedural macros (case macros) [20] that enable true composability. In addition there has been work to integrate procedural macros and module systems [13, 17]. Racket takes it even further by extending the Scheme macro system with deep hooks into the compilation process [14, 35] and robust pattern specifications [9].

In addition, there are a number of macro systems for languages with more traditional syntax that are not fully delimited. As mentioned before, sweet.js is most closely related to Honu [29, 30]. In contrast with Honu, which does not include regular expression literals, we solve the reader ambiguity problem for JavaScript and introduce infix macros along with the `invoke` pattern class.

Macro systems that use a similar technique as Honu include Fortress [4] and Dylan [5] however they only provide support for term rewriting macros (our `rule` macros). Dylan's auxilary rules are similar to our `invoke` pattern class. Nemerle [33] also uses a similar technique but does not allow local definitions of macros.

The Marco system [23] is an interesting alternative approach that presents a cross-language macro system. Rather than tightly integrate the macro system with a specific language Marco provides a separate macro definition language that can compile to multiple languages. While this approach provides generality it sacrifices language specific expressiveness (e.g. name clashes are errors in their system while they are just renamed in ours).

C++ templates [3] are a powerful compile time meta programming facility for C++. In contrast to C++ templates sweet.js provides more syntactic flexibility in the definable syntactic forms along with the ability to define transformations in the host language rather than the just the template language.

Template Haskell [32] makes a tradeoff by forcing the macro call sites to always be demarcated. The means that macros are always a second class citizen; macros in Haskell cannot seamlessly build a language on top of Haskell in the same way that Scheme and Racket can.

Some systems such as SugarJ [11], ometa [37], Xtc [18], Xoc [8], and Polyglot [26] provide extensible grammars but require the programmer to reason about parser details. Multi stage systems such as mython [31] and MetaML [25, 34] can also be used to create macros systems like MacroML [16]. Systems like Stratego [36] transforms syntax using its own language, separate from the host language. Metaborg [6] and SugarJ [11] are syntax extension facilities built on top of Stratego.

8. Conclusion

We have presented sweet.js, a hygienic macro system for JavaScript. Our macro system follows in the footsteps of Scheme by providing declarative and procedural macro definition functionality.

In addition we presented an algorithm for read that correctly handles key ambiguities in the JavaScript grammar that allows the lexer and parser to be separated for the first time in JavaScript.

References

[1] The Esprima JavaScript Parser. `http://esprima.org/`.

[2] The Rust Language. `http://www.rust-lang.org/`.

[3] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. 2001.

[4] E. Allen, R. Culpepper, and J. Nielsen. Growing a syntax. *Proceedings of Workshop on Foundations of Object-Oriented Languages*, 2009.

[5] J. Bachrach, K. Playford, and C. Street. D-Expressions : Lisp Power , Dylan Style. *Style DeKalb IL*, 1999.

[6] M. Bravenboer and E. Visser. Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. *OOPSLA '04 Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2004.

[7] W. Clinger. Macros that work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '91*, pages 155–162, New York, New York, USA, Jan. 1991. ACM Press.

[8] R. Cox, T. Bergan, and A. Clements. Xoc, an extension-oriented compiler for systems programming. *ACM SIGARCH Computer Architecture News*, 2008.

[9] R. Culpepper and M. Felleisen. Fortifying macros. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming - ICFP '10*, volume 45, page 235, New York, New York, USA, Sept. 2010. ACM Press.

[10] T. Disney, N. Faubion, D. Herman, and C. Flanagan. The Sweet.js Appendix. `https://github.com/mozilla/sweet.js/blob/paper/doc/paper/sweetjs-appendix.pdf`.

[11] S. Erdweg, T. Rendel, C. Kästner, and K. Ostermann. SugarJ: library-based syntactic language extensibility. *OOPSLA '11 Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, 2011.

[12] N. Faubion. The Sparkler project. `https://github.com/natefaubion/sparkler`.

[13] M. Flatt. Composable and compilable macros: You Want it When? *ICFP '02 Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, 37(9):72–83, Sept. 2002.

[14] M. Flatt and R. Culpepper. Macros that Work Together. *Journal of Functional Programming*, 2012.

[15] J. Foderaro, K. Sklower, and K. Layer. *The FRANZ Lisp Manual*. 1983.

[16] S. Ganz, A. Sabry, and W. Taha. Macros as multi-stage computations: type-safe, generative, binding macros in MacroML. *ICFP '01 Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, 2001.

[17] A. Ghuloum and R. K. Dybvig. Implicit phasing for R6RS libraries. *ICFP '07 Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, 42(9):303, Oct. 2007.

[18] R. Grimm. Better extensibility through modular syntax. *PLDI '06 Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, 2006.

[19] S. P. Harbison and J. Steele, G. L. *C: A Reference Manual*. Prentice-Hall, 1984.

[20] R. Hieb, R. Dybvig, and C. Bruggeman. Syntactic abstraction in scheme. *Lisp and symbolic computation*, 5(4):295–326, 1992.

[21] E. C. M. A. International. *ECMA-262 ECMAScript Language Specification*. Number June. ECMA (European Association for Standardizing Information and Communication Systems), 5.1 edition, 2011.

[22] E. E. Kohlbecker and M. Wand. Macro-by-example: Deriving syntactic transformations from their specifications. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '87*, pages 77–84, New York, New York, USA, Oct. 1987. ACM Press.

[23] B. Lee, R. Grimm, M. Hirzel, and K. McKinley. Marco: safe, expressive macros for any language. *ECOOP 2012–Object-Oriented . . .*, 2012.

[24] J. Long. The es6-macros project. `https://github.com/jlongster/es6-macros`.

[25] M. Martel and T. Sheard. Introduction to multi-stage programming using metaml. 1997.

[26] N. Nystrom, M. Clarkson, and A. Myers. Polyglot: An extensible compiler framework for Java. *Compiler Construction*, 2003.

[27] PerlMonks. On Parsing Perl. `http://www.perlmonks.org/?node_id=44722`.

[28] K. M. Pitman. Special forms in Lisp. In *Proceedings of the 1980 ACM conference on LISP and functional programming - LFP '80*, pages 179–187, New York, New York, USA, Aug. 1980. ACM Press.

[29] J. Rafkind. *Syntactic extension for languages with implicitly delimited and infix syntax*. PhD thesis, 2013.

[30] J. Rafkind and M. Flatt. Honu: Syntactic Extension for Algebraic Notation through Enforestation. *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, 2012.

[31] J. Riehl. Language embedding and optimization in mython. *DLS '09 Proceedings of the 5th symposium on Dynamic languages*, 2009.

[32] T. Sheard and S. Jones. Template meta-programming for Haskell. *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, 2002.

[33] K. Skalski, M. Moskal, and P. Olszta. Meta-programming in Nemerle. *Proceedings Generative Programming and Component Engineering*, 2004.

[34] W. Taha and T. Sheard. Multi-stage programming with explicit annotations. *PEPM '97 Proceedings of the 1997 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, 1997.

[35] S. Tobin-Hochstadt and V. St-Amour. Languages as libraries. *PLDI '11 Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011.

[36] E. Visser. Program transformation with Stratego/XT. *Domain-Specific Program Generation*, 2004.

[37] A. Warth and I. Piumarta. OMeta: an object-oriented language for pattern matching. *Proceedings of the 2007 symposium on Dynamic languages*, 2007.