Sweet.js - Hygienic Macros for JavaScript

Tim Disney
UC Santa Cruz

Nate Faubion

David Herman
Mozilla

Cormac Flanagan
UC Santa Cruz

Abstract

Sweet.js is a hygienic macro system for JavaScript.

1. Introduction

Macros systems have a long history in the design of extensible programming languages going back at least to Lisp and Scheme [7, 4] as a tool to provide programmers syntactic flexibility.

While powerful macro systems have been used extensively in Lisp-derived languages, there has been considerable less movement for macros systems for languages with an

expression based syntax such as JavaScript. This is due to a variety of technical reasons that have held back macro systems in expression based languages which we address in this paper.

Key ideas: Parens in Scheme make macros easy. Rich syntax is much harder. Need to interpose reader before parsing. ES5's parser uses context-sensitive lexing. Need a reader that avoids context sensitive lexing and produces token trees.

A scheme compiler pipeline looks like:

$$\mathit{Scheme} : \mathit{lexer} \xrightarrow{\mathit{Token}^*} \mathit{reader} \xrightarrow{\mathit{Sexpr}} \mathit{parser} \xrightarrow{\mathit{AST}}$$

A because of ambiguity during lexing an ES5 compiler pipline looks like:

$$\mathit{ES5} : \mathit{lexer} \underset{\mathit{Token}^*}{\overset{\mathit{feedback}}{\rightleftharpoons}} \mathit{parser} \xrightarrow{\mathit{AST}}$$

And a sweet.js pipline is

$$\mathit{sweet.js} : \mathit{lexer} \xrightarrow{\mathit{Token}^*} \mathit{reader} \xrightarrow{\mathit{TokenTree}^*} \mathit{parser} \xrightarrow{\mathit{AST}}$$

Our website is at [2].

Recently the Honu project [9, 8] has shown how to overcome some of the existing challenges in developing a macro system for expression based language. The Honu technique was designed for an idealized JavaScript like language.

In this paper we present sweet.js, a hygienic macros system for JavaScript, that extends the techniques of Honu for full JavaScript and present additional techniques that target expression based languages.

We make the following contributions:

- an implementation of and proof of *read* that addresses the JavaScript grammar issues that traditionally have prevented separating lexing and parsing.
- the ability to define infix macros that match on arbitrary surrounding syntax.
- the `invoke` primitive to allow custom parser classes and more powerful matching

2. Overview

The sweet.js system provides two kinds of macros: *rule* macros (analogous to `syntax-rules` in Scheme) and *case* macros (analogous to `syntax-case` in Scheme). 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 (. . .) mean match zero or more tokens.

The above example show 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 match on an expression so in this example the entire expression `40 + 2` is bound to `$init`.

(some better examples than def and var?)

Along with the template-based rule macros, sweet.js also provides the more powerful case macros. Instead of a template, the body of a case macro is JavaScript that is run when the macro is invoked.

```
macro log {
  case {_ ($msg) } => {
    // need a good example here...
  }
}
```

3. Enforestation

The core technique that enables Scheme-like expressive macros for non delimited languages is called *enforestation* [9]. Enforestation works by progressively recognizing syntax forms and expanding macros. The key idea of enforestation is to give structure to the syntax arguments to macros.

Sweet.js implements this algorithm mostly as described with some additions to provide infix macros and invoke pattern classes (described in the following sections).

Enforestation works by grouping token trees into a more structured form called a term tree. Term trees are a kind of proto-AST; each term tree corresponds to a node in the final AST. For example,

```
2 + 40;
foo(100)
```

After an initial pass through the enforestation process the first binary expression will be matched as a binary term tree (we use angle brackets to denote a term tree element).

```
<binop: +, left: 2, right: 40>;
foo(100)
```

Through enforestation, a macro can match structured groupings of code. For example, consider a `let` macro that must match the variable initialization (which can be an arbitrary undelimited expression):

```
macro let {
  rule { $name = $init:expr } => {
    // ...
  }
}
let x = 40 + 2;
```

The `:expr` pattern class signals the expander to match an expression term tree, which in the case is `<binop +, left: 40, right: 2>`.

3.1 Infix Macros

The macros we have described so far must all be prefixed by the macro identifier and syntax after the macro name is matched. This is sufficient for many kinds of macros but some syntax forms require the macro identifier to sit between patterns.

Honu addresses this need in a limited way by providing a way to define new binary and unary operators which during expansion can manipulate their operators. However, those operators must be fully expanded and must match as an expression.

Sweet.js provides *infix macros* which allows a macro identifier to match syntax before it. For example, the following implements ES6-style arrow functions via infix macros:

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

var id = (x) => { return x; }
```

This is accomplished by simply providing the state of previously expanded syntax to macro transformers. Macros may then consume from either ends as needed, yielding new previous and subsequent syntax. At first glance, this can appear brittle:

```
var foo = bar(x) => { return x; }
```

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

```
var foo = bar function(x) { return x; }
```

In more subtle cases, a naive expansion can result in parsable code with incorrect semantics. [Example needed?] To preserve the integrity of previously expanded syntax, we verify that an infix macro only matches previous syntax on boundaries delimited by the partial syntax tree we've built. The syntax tree would show `bar(x)` as a complete function call term. Consuming the parentheses would result in a split term and is disallowed, failing the rule. In practice, this has proven to be an intuitive restriction. [Elaborate?]

The primary limitation of infix macros is they do not obey precedence and associativity. They stand as a complement to Honu operators, not as a replacement, for when an infix form does not adhere to operator semantics, much like the `=>` macro. Its left-hand-side and right-hand-side are not arbitrary expressions but must match a specific form. Additionally, it would need a precedence higher than other operators.

3.2 Invoke and Pattern Classes

[Something about default pattern classes and ':expr'...]

Pattern classes are extensible via the `invoke` class which is parameterized by a macro name.

```
macro color {
  rule { red } => { red }
  rule { green } => { green }
  rule { blue } => { blue }
}
macro colors_options {
  rule { ($opt:invoke(color) ...) } => {
      ... }
}
```

The macro is essentially inserted into the token stream. If the expansion succeeds, the result will be loaded into the pattern variable, otherwise the rule will fail. We've added sugar so that any non-primitive pattern classes are interpretted as `invoke` parameterized by the custom class. We've also added identity rules to shorten definitions of simple custom classes.

```
macro color {
  rule { red }
  rule { green }
  rule { blue }
}
macro colors_options {
  rule { ($opt:color (,) ...) } => { ... }
}
```

These macros may return an optional pattern environment which will be scoped and loaded into the invoking macro's pattern environment. This lets us define Honu-style pattern classes as simple macro-generating macros.

```
macro color {
  ...
}
macro number {
  ...
}
// ‘pattern‘ is just a macro-generating
    macro
pattern color_value { $color:color $num:
    number }

macro color_options {
  rule { ($opt:color_value (,) ...) } => {
    var cols = [$opt$color (,) ...];
    var nums = [$opt$num (,) ...];
  }
}
```

...?

4. 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 macro system must transform code before it reaches the parser, but in order to implement expressive macros it requires more structured input than a simple token stream.

In Lisp and Scheme, this additional structure is provided by the read function, that transforms a token stream into an s-expression by matching delimiters. By operating on the s-expression representation, macros can manipulate whole delimited chunks of code.

While this is obviously a requirement in an fully delimited language like Lisp and Scheme, it is also important in languages such as JavaScript that are not entirely delimited.

TODO: class example to show the delimiters?

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.

Our macro system includes a reader that transforms a stream of flat tokens into a stream of more structured *token trees*, by matching delimiters.

Unfortunately, the syntax of JavaScript presents a challenge to correctly implement the `read` function. In particular, one of the syntax features of JavaScript is support for regular expression literals, which are formed with the forward slash `/`. Delimiters are valid inside of a regular expression literal so the reader must not match delimiters inside of a regular expression literal otherwise it would break the true delimiter structure of the program.

```
function makeRegex() {
  return /}/;
}
```

Since the reader must ignore delimiters inside regular expression literals, it must know when it is reading between the literal's slashes. However, slash is also used as the divide operator in JavaScript. So, in order for the reader to ignore delimiters inside of regex literals it must first decide if a slash is the beginning of a literal or a divide operator. Unfortunately this is somewhat complicated for JavaScript.

Traditionally, JavaScript parsers are not separated from the lexer; the parser calls the lexer with a flag indicating if the grammatical context accepts either a regular expression literal or a divide token. For JavaScript parsers this technique is sufficient but a macro expander requires the separation of the lexer and reader from the parser.

Note that this problem is not present in Honu since their language does not include regular expression literals.

A key novelty in sweet.js is the design and implementation of a correct version of read for full ES5 JavaScript[1]. For clarity of presentation, this paper describes the implementation of read for the subset of

[1] Our implementation also has initial support for the upcoming ES6 version of JavaScript

JavaScript shown in Figure 3 which retains the essential complications of a correct version of read but elides the full details of the entire JavaScript language.

Read takes a sequence of tokens and produces a sequence of token trees. Tokens are punctuators (`+`, `/`, `:`, `;`, `=`, `.`), keywords (`return`, `function`, `if`), variables (x), and delimiters (`{`, `(`, `}`, `)`). We write a sequence of tokens separated by a dot so the source string "`foo(/x/)`" is lexed into the sequence of six tokens `foo`·`(`·`/`·`x`·`/`·`)`.

A token tree is similar to a token with the addition of the regular expression literal (/x/) and instead of individual opening and closing delimiter tokens a delimiter nests a sequence of inner token trees (e.g. the sequence `foo`·`(`·`/`·`x`·`/`·`)` would be the token tree `foo`·`(`/x/`)`).

The key idea of read is to maintain of 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.

Some of the cases of read are relatively obvious. For example, if the token just read was one of the binary operators (e.g. `100 + /x/g`) the slash will always be a regular expression literal. If on the other hand the previous token tree was a parenthesis then most of the time the slash will be the divide operator (e.g. dividing the result of a call `foo(100)/ 10`). However, if the token before the parenthesis was the keyword `if` then it is actually the start of a regular expression (since single statement if bodies do not require braces).

```
if (x) /y/g
```

One of the most complicated cases is a slash following curly braces. Part of the complication here is that curly braces can be either an object literal (in which case the slash should be a divide) or it could be 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:

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

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.

But if we slightly change the code the outer curly braces become an object literal and `x` is a property so

Figure 1: AST for Simplified JavaScript

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

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 block 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.

4.1 Proving Read

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

The parser for normal tokens is defined in Figure 3. A parser for the nonterminal *Program* is a function from a sequence of tokens to an AST.

$$\mathit{Program} :: \mathit{Token}^* \rightarrow \mathit{AST}$$

We use the notation $\mathit{Program}_e ::= \mathit{SourceElements}_e$ to mean match the input sequence with $\mathit{SourceElements}_e$ and produce the resulting AST e.

Note that the grammar we present here is a simplified version of the grammar specified in the ECMAScript standard [6] and many of the nonterminal names we use here correspond to shortened versions of nonterminals in the standard.

The language presented here is a simplified for the sake of presentation; it is mostly straightforward to extend the algorithm presented here for the simplified language to the sweet.js implementation for full 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 similar except for that all rules with delimiters and regular expression literals will change:

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

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

- The traditional parsing strategy is, given a token stream s, to parse s into an AST e using a traditional parser.
- The second parsing strategy first reads s into a token tree stream $t = \text{read}(s,\ \epsilon,\ \textit{false})$, and then parses this token tree stream t into an AST e.

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

Proof. By showing parse equivalence for each non-terminal in the grammar. Details in the appendix. □

5. Hygiene

The main hygienic technique for procedurual macros were described in [5]. Earlier work for syntax-rules is [1].

Mostly straightforward implementation from scheme (in for a good treatment of the subject see [3]) with some details to handle `var`.

5.1 Var statements

Unfortunately there's some complexity in how `var` (and `let`/`const`) statements and macros interact. In short, macros must be defined before being used in a var statement.

For example, this will work:

```
macro id { rule { $id } }
var x = id 42;
// -->
var x = 42;
```

Figure 2: Simplified Read Algorithm

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

$$
\begin{array}{lcll}
\multicolumn{4}{l}{\text{isExprPrefix} : \textit{TokenTree}^{*} \to \textit{Bool} \to \textit{Int}} \\
\text{isExprPrefix}(\epsilon,\ \textit{true},\ l) & = & \textit{true} & \\
\text{isExprPrefix}(p \cdot \texttt{/},\ b,\ l) & = & \textit{true} & \\
\text{isExprPrefix}(p \cdot \texttt{+},\ b,\ l) & = & \textit{true} & \\
\text{isExprPrefix}(p \cdot \texttt{=},\ b,\ l) & = & \textit{true} & \\
\text{isExprPrefix}(p \cdot \texttt{:},\ b,\ l) & = & b & \\
\text{isExprPrefix}(p \cdot \texttt{return}^{l},\ b,\ l') & = & \textit{false} & \textit{if}\ l \neq l' \\
\text{isExprPrefix}(p \cdot \texttt{return}^{l},\ b,\ l') & = & \textit{true} & \textit{if}\ l = l' \\
\text{isExprPrefix}(p,\ b,\ l) & = & \textit{false} &
\end{array}
$$

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\text{read} : \textit{Token}^{*} \to \textit{TokenTree}^{*} \to \textit{Bool} \to \textit{TokenTree}^{*}} \\
\text{read}(\texttt{/} \cdot x \cdot \texttt{/} \cdot s,\ \epsilon,\ b) & = & \texttt{/}x\texttt{/} \cdot \text{read}(s,\ \texttt{/}x\texttt{/},\ b) \\
\text{read}(\texttt{/} \cdot x \cdot \texttt{/} \cdot s,\ p \cdot p',\ b) & = & \texttt{/}x\texttt{/} \cdot \text{read}(s,\ p \cdot p' \cdot \texttt{/}x\texttt{/},\ b) \\
\quad \textit{if}\ p' \in \textit{Punctuator or Keyword} & & \\
\text{read}(\texttt{/} \cdot x \cdot \texttt{/} \cdot s,\ p \cdot \texttt{if} \cdot \texttt{(}t\texttt{)},\ b) & = & \texttt{/}x\texttt{/} \cdot \text{read}(s,\ p \cdot \texttt{if} \cdot \texttt{(}t\texttt{)} \cdot \texttt{/}x\texttt{/},\ b) \\
\text{read}(\texttt{/} \cdot x \cdot \texttt{/} \cdot s,\ p \cdot p' \cdot \texttt{function}^{l} \cdot x \cdot \texttt{(}t\texttt{)} \cdot \texttt{\{}t'\texttt{\}},\ b) & = & \texttt{/}x\texttt{/} \cdot \text{read}(s,\ p \cdot p' \cdot \texttt{function}^{l} \cdot x \cdot \texttt{(}t\texttt{)} \cdot \texttt{\{}t'\texttt{\}} \cdot \texttt{/}x\texttt{/},\ b) \\
\quad \textit{if false} \ = \ \text{isExprPrefix}(p',\ b,\ l) & & \\
\text{read}(\texttt{/} \cdot x \cdot \texttt{/} \cdot s,\ p \cdot \texttt{\{}t\texttt{\}}^{l},\ b) & = & \texttt{/}x\texttt{/} \cdot \text{read}(s,\ p \cdot \texttt{\{}t\texttt{\}}^{l} \cdot \texttt{/}x\texttt{/},\ b) \\
\quad \textit{where false} = \text{isExprPrefix}(p,\ b,\ l) & & \\
 & & \\
\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 \texttt{(}t\texttt{)},\ b) & = & \texttt{/} \cdot \text{read}(s,\ p \cdot \texttt{(}t\texttt{)} \cdot \texttt{/},\ b) \\
\text{read}(\texttt{/} \cdot s,\ p \cdot p' \cdot \texttt{function}^{l} \cdot x \cdot \texttt{(}t\texttt{)} \cdot \texttt{\{}t'\texttt{\}},\ b) & = & \texttt{/} \cdot \text{read}(s,\ p \cdot p' \cdot \texttt{function}^{l} \cdot x \cdot \texttt{(}t\texttt{)} \cdot \texttt{\{}t'\texttt{\}} \cdot \texttt{/},\ b) \\
\quad \textit{if true} \ = \ \text{isExprPrefix}(p',\ b,\ l) & & \\
\text{read}(\texttt{/} \cdot s,\ p \cdot \texttt{\{}t\texttt{\}}^{l},\ b) & = & \texttt{/} \cdot \text{read}(s,\ p \cdot \texttt{\{}t\texttt{\}}^{l} \cdot \texttt{/},\ b) \\
\quad \textit{where true} = \text{isExprPrefix}(p,\ b,\ l) & & \\
 & & \\
\text{read}(\texttt{(} \cdot s \cdot \texttt{)} \cdot s',\ p,\ b) & = & \texttt{(}t\texttt{)} \cdot \text{read}(s',\ p \cdot \texttt{(}t\texttt{)},\ b) \\
\quad \textit{where } s \textit{ contains no unmatched } \texttt{)} & & \quad \textit{where}\ t = \text{read}(s,\ \epsilon,\ \textit{false}) \\
\text{read}(\texttt{\{}^{l} \cdot s \cdot \texttt{\}} \cdot s',\ p,\ b) & = & \texttt{\{}t\texttt{\}}^{l} \cdot \text{read}(s',\ p \cdot \texttt{\{}t\texttt{\}}^{l},\ b) \\
\quad \textit{where } s \textit{ contains no unmatched } \texttt{\}} & & \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}
$$

Figure 3: Simplified ES Grammar

$\mathit{PrimaryExpr}_x$	$::=$	x
$\mathit{PrimaryExpr}_{/x/}$	$::=$	$/ \cdot x \cdot /$
$\mathit{PrimaryExpr}_{\{\texttt{x}:e\}}$	$::=$	$\{\texttt{x}:\mathit{AssignExpr}_e\}$
$\mathit{PrimaryExpr}_{(e)}$	$::=$	$(\mathit{AssignExpr}_e)$
$\mathit{MemberExpr}_e$	$::=$	$\mathit{PrimaryExpr}_e$
$\mathit{MemberExpr}_e$	$::=$	$\mathit{FunctionExpr}_e$
$\mathit{MemberExpr}_{e.\texttt{x}}$	$::=$	$\mathit{MemberExpr}_e\ .\ \texttt{x}$
$\mathit{CallExpr}_{e\ (e')}$	$::=$	$\mathit{MemberExpr}_e\ (\mathit{AssignExpr}_{e'})$
$\mathit{CallExpr}_{e\ (e')}$	$::=$	$\mathit{CallExpr}_e\ (\mathit{AssignExpr}_{e'})$
$\mathit{CallExpr}_{e.\texttt{x}}$	$::=$	$\mathit{CallExpr}_e\ .\ \texttt{x}$
$\mathit{BinaryExpr}_e$	$::=$	$\mathit{CallExpr}_e$
$\mathit{BinaryExpr}_{e\ /\ e'}$	$::=$	$\mathit{BinaryExpr}_e\ /\ \mathit{BinaryExpr}_{e'}$
$\mathit{BinaryExpr}_{e\ +\ e'}$	$::=$	$\mathit{BinaryExpr}_e + \mathit{BinaryExpr}_{e'}$
$\mathit{AssignExpr}_e$	$::=$	$\mathit{BinaryExpr}_e$
$\mathit{AssignExpr}_{e\ =\ e'}$	$::=$	$\mathit{CallExpr}_e = \mathit{AssignExpr}_{e'}$
$\mathit{StmtList}_e$	$::=$	Stmt_e
$\mathit{StmtList}_{e\ e'}$	$::=$	$\mathit{StmtList}_e\ \mathit{Stmt}_{e'}$
$\mathit{Stmt}_{\{e\}}$	$::=$	$\{\mathit{StmtList}_e\}$
$\mathit{Stmt}_{\texttt{x}:\ e}$	$::=$	$\texttt{x}\ :\ \mathit{Stmt}_e$
Stmt_e	$::=$	$\mathit{AssignExpr}_e$; *where lookahead* $\neq$ { *or* `function`
$\mathit{Stmt}_{\texttt{if}\ (e)\ e'}$	$::=$	$\texttt{if}\ (\mathit{AssignExpr}_e)\ \mathit{Stmt}_{e'}$
$\mathit{Stmt}_{\texttt{return}}$	$::=$	`return`
$\mathit{Stmt}_{\texttt{return}\ e}$	$::=$	`return` [no line terminator here] $\mathit{AssignExpr}_e$;
$\mathit{FunctionDecl}_{\texttt{function}\ x\ (x')\ \{e\}}$	$::=$	$\texttt{function}\ x\ (x')\ \{\mathit{SourceElements}_e\}$
$\mathit{FunctionExpr}_{\texttt{function}\ x\ (x')\ \{e\}}$	$::=$	$\texttt{function}\ x\ (x')\ \{\mathit{SourceElements}_e\}$
$\mathit{SourceElement}_e$	$::=$	Stmt_e
$\mathit{SourceElement}_e$	$::=$	$\mathit{FunctionDecl}_e$
$\mathit{SourceElements}_e$	$::=$	$\mathit{SourceElement}_e$
$\mathit{SourceElements}_{e\ e'}$	$::=$	$\mathit{SourceElements}_e\ \mathit{SourceElement}_{e'}$
$\mathit{Program}_e$	$::=$	$\mathit{SourceElements}_e$
$\mathit{Program}$	$::=$	ϵ

But not this:

```
var x = id 42;
macro id { rule { $id } }
// -->
var x = m 42;
```

This is unfortunate because macros can be used before definition for all other syntax forms, just not `var`. Also Racket, from which sweet.js takes most other design cues, allows exactly this kind of use/definition mixing with internal definitions:

```
(define y (id 100))
(define-syntax-rule (id x) x)
```

In order to support mixing use/def the scheme expander must run through each scope twice. The first pass registers any macro definitions but critically it doesn't expand inside internal definitions.

So after the first pass of the above example we will have something like:

```
// the id macro is registered in the
   environment { id: <transformer> }
(define y (id 100))
```

And then the second pass will expand macros inside internal definitions:

```
(define y 100)
```

So far so good, but the tricky part for sweet.js is you need to know where internal definitions end during the first pass *before* expanding any macros inside them. This is easy for Scheme since everything is delimited but this is not the case in JavaScript.

To clarify the problem, consider the ES6 arrow macro:

```
var x = (x) => { x }
macro => { ... }
```

When the expander gets to the `var` statement during the first pass it doesn't know anything about the `=>` macro yet. All it can know is that the right hand side of `=` needs to be an expression (which in this case means it will match `var x = (x)` and then fail to match on `=>`).

The basic problem here is that in JavaScript there is no way to detect the end of a var statement in the presence of as yet unknown macro definitions. So, you must define your macros before using them in `var` statements.

So even though we cannot defer expansion inside of `var` statements we still do two passes for each scope and the expansion of any syntax inside of a delimiter is deferred until the second pass which now has all the macro definitions in scope. Which means that you can actually use macros in `var` statements before definition as long as you wrap it in parens.

```
var x = id(100);
// var x = id(100)

var x = (id(100));
// var x = 100

macro id { rule { ($x) } }
```

6. Implementation

Sweet.js is written in JavaScript and runs in the major JS environments (i.e. the brower and node.js). This is in contrast to Honu which translates its code to Racket code and reuses the hygienic expansion machinery already built in Racket. While this simplifies the implementation of Honu it also requires an installation of Racket which in some cases is not feasible (e.g. sweet.js is able to run in mobile device browsers).

7. Related Work

Earlier treatments of macro-by-example in [7].

- Scheme/Racket
- Honu
- Template Haskell
- Nemerle [10]
- Scala
- Closure

8. Conclusion

References

[1] William 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, January 1991. ACM Press.

[2] Tim Disney. Sweet.js Project, 2014.

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

[4] JK Foderaro, KL Sklower, and K Layer. *The FRANZ Lisp Manual*. 1983.

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

[6] 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.

[7] 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, October 1987. ACM Press.

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

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

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

A. Read Proof

Theorem 2 (Parse Equivalence for Program)**.**
$\forall s.$

$$\begin{array}{rl} & s \in \mathit{Program}_e \\ \Leftrightarrow & \mathit{read}(s,\ \epsilon,\ \mathit{false}) \in \mathit{Program'}_e \end{array}$$

Proof. For the left-to-right direction, the are two production rules for $\mathit{Program}_e$.

- $s \in \epsilon$. The result is immediate.
- $s \in \mathit{SourceElements}_e$ this holds by Lemma 1.

A similar argument holds for the right-to-left direction. □

Lemma 1 (Parse Equivalence for SourceElements)**.**
$\forall s.$

$$\begin{array}{rl} & s \in \mathit{SourceElements}_e \\ \Leftrightarrow & \mathit{read}(s,\ \epsilon,\ \mathit{false}) \in \mathit{SourceElements'}_e \end{array}$$

Proof. For the left-to-right direction there are two production rules for $\mathit{SourceElements}_e$.

- $s \in \mathit{SourceElement}_e$. This holds by Lemma 2.
- $s \in \mathit{SourceElements}_e\ \mathit{SourceElement}_{e'}$. We have $s = s' \cdot s''$ where $s' \in \mathit{SourceElements}_e$ and $s'' \in \mathit{SourceElement}_{e'}$.

$$\begin{array}{rcl} t & = & \mathrm{read}(s' \cdot s'',\ \epsilon,\ \mathit{false}) \\ & = & \mathrm{read}(s',\ \epsilon,\ \mathit{false}) \cdot \mathrm{read}(s'',\ \mathrm{read}(s',\ \epsilon,\ \mathit{false}),\ \mathit{false}) \end{array}$$

By induction $\mathrm{read}(s',\ \epsilon,\ \mathit{false}) \in \mathit{SourceElements'}_e$ and by Lemma 2, $\mathrm{read}(s'',\ \mathrm{read}(s',\ \epsilon,\ \mathit{false}),\ \mathit{false}) \in \mathit{SourceElement'}_{e'}$ (since by Lemma 12, $\mathrm{read}(s',\ \epsilon,\ \mathit{false}) \in \mathit{RegexPrefix}$). Thus $t \in \mathit{SourceElements'}_{e\ e'}$.

The argument is similar for the right-to-left direction. □

Lemma 2 (Parse Equivalence for SourceElement)**.**
$\forall s, p \in \mathit{RegexPrefix}.$

$$\begin{array}{rl} & s \in \mathit{SourceElement}_e \\ \Leftrightarrow & \mathit{read}(s,\ p,\ \mathit{false}) \in \mathit{SourceElement'}_e \end{array}$$

Proof. For the left-to-right direction there are two production rules for $\mathit{SourceElement}_e$.

- $s \in \mathit{Stmt}_e$. This holds by Lemma 4 since $p \in \mathit{RegexPrefix}$.
- $s \in \mathit{FunctionDecl}_e$. This holds by Lemma 3.

The argument is similar for the right-to-left direction.

□

Lemma 3 (Parse Equivalence for FunctionDecl)**.**
$\forall s, p, b.$

$$\begin{array}{rl} & s \in \textit{FunctionDecl}_e \\ \Leftrightarrow & \textit{read}(s,\ p,\ b) \in \textit{FunctionDecl'}_e \end{array}$$

Proof. For the left-to-right direction there is one production rule for *FunctionDecl*$_e$:

$$s \in \texttt{function}\ x\ \texttt{(}x'\texttt{)}\ \texttt{\{}\textit{SourceElements}_e\texttt{\}}$$

We have $s = \texttt{function}\ x\ \texttt{(}x\texttt{)}\ \texttt{\{}s'\texttt{\}}$ where $s' \in$ *SourceElements*$_e$ so:

$$\begin{array}{rcl} t & = & \text{read}(\texttt{function}\ x\ \texttt{(}x\texttt{)}\ \texttt{\{}s'\texttt{\}},\ p,\ b) \\ & = & \texttt{function}\ x\ \texttt{(}x\texttt{)}\ \texttt{\{}t'\texttt{\}} \end{array}$$

where $t' = \text{read}(s',\ \epsilon,\ \textit{false})$. Since by Lemma 1, $t' \in$ *SourceElements'*$_e$ we have $t \in \textit{FunctionDecl'}_{\texttt{function}\ x\ \texttt{(}x\texttt{)}\ \texttt{\{}e\texttt{\}}}$.
The argument is similar for the right-to-left direction.

□

Lemma 4 (Parse Equivalence for Stmt)**.**
$\forall s, p \in \textit{RegexPrefix}.$

$$\begin{array}{rl} & s \in \textit{Stmt}_e \\ \Leftrightarrow & \textit{read}(s,\ p,\ \textit{false}) \in \textit{Stmt'}_e \end{array}$$

Proof. For the left-to-right direction we have several production rules for *Stmt*$_e$.

- $s \in \texttt{\{}\textit{StmtList}_e\texttt{\}}$. We have $s = \texttt{\{}s'\texttt{\}}$ where $s' \in$ *StmtList*$_e$. Then:

$$\begin{array}{rcl} t & = & \text{read}(\texttt{\{}s'\texttt{\}},\ p,\ \textit{false}) \\ & = & \texttt{\{}t'\texttt{\}} \end{array}$$

  where $t' = \text{read}(s',\ \epsilon,\ \textit{false})$. By Lemma 5, $t' \in$ *StmtList'*$_e$ so $t \in \textit{Stmt'}_{\texttt{\{}e\texttt{\}}}$.
- $s \in \textit{AssignExpr}_e\ \texttt{;}$. We have $s = s' \cdot \texttt{;}$ where $s' \in \textit{AssignExpr}_e$. Then:

$$\begin{array}{rcl} t & = & \text{read}(s' \cdot \texttt{;},\ p,\ \textit{false}) \\ & = & \text{read}(s',\ p,\ \textit{false}) \cdot \texttt{;} \end{array}$$

  Since $p \in \textit{RegexPrefix}$ by Lemma 6, $\text{read}(s',\ p,\ \textit{false}) \in$ *AssignExpr'*$_e$ we have $t \in \textit{Stmt'}_e$.

- $s \in$ `if` `(`*AssignExpr*$_e$`)` *Stmt*$_{e'}$. We have $s = \texttt{if} \cdot \texttt{(}s'\texttt{)} \cdot s''$ where $s' \in \mathit{AssignExpr}_e$ and $s'' \in \mathit{Stmt}_{e'}$.

$$\begin{array}{rcl} t & = & \mathrm{read}(\texttt{if} \cdot \texttt{(}s'\texttt{)} \cdot s'',\ p,\ \mathit{false}) \\ & = & \texttt{if} \cdot \texttt{(}t'\texttt{)} \cdot t'' \end{array}$$

  where

$$\begin{array}{rcl} t' & = & \mathrm{read}(s',\ \epsilon,\ \mathit{false}) \\ t'' & = & \mathrm{read}(s'',\ p \cdot \texttt{if} \cdot \texttt{(}t'\texttt{)},\ \mathit{false}) \end{array}$$

  By Lemma 6 $t' \in$ *AssignExpr'*$_e$. By induction, $t'' \in$ *Stmt'*$_{e'}$ (since $p \cdot \texttt{(}t'\texttt{)} \in$ *RegexPrefix*). Thus $t \in$ *Stmt'*$_{\texttt{if (}e\texttt{)}\ e'}$.
- $s \in$ `return`. Since $s =$ `return` and $\mathrm{read}(s,\ p,\ \mathit{false}) =$ `return` $\in$ *Stmt'*$_{\texttt{return}}$ we have our result directly.
- $s \in$ `return` *AssignExpr*$_e$`;`. We have $s = \texttt{return} \cdot s' \cdot \texttt{;}$ where $s' \in \mathit{AssignExpr}_e$. Then:

$$\begin{array}{rcl} t & = & \mathrm{read}(\texttt{return} \cdot s' \cdot \texttt{;},\ p,\ \mathit{false}) \\ & = & \texttt{return} \cdot \mathrm{read}(s',\ p \cdot \texttt{return},\ \mathit{false}) \cdot \texttt{;} \end{array}$$

  By Lemma 6, $\mathrm{read}(s',\ p \cdot \texttt{return},\ \mathit{false}) \in$ *AssignExpr'*$_e$ thus $t \in$ *Stmt'*$_{\texttt{return}\ e}$.
- $s \in \texttt{x} : \mathit{Stmt}_e$. We have $s = \texttt{x} \cdot \texttt{:} \cdot s'$ where $s' \in \mathit{Stmt}_e$. Then:

$$\begin{array}{rcl} t & = & \mathrm{read}(\texttt{x} \cdot \texttt{:} \cdot s',\ p,\ \mathit{false}) \\ & = & \texttt{x} \cdot \texttt{:} \cdot \mathrm{read}(s',\ p \cdot \texttt{x} \cdot \texttt{:},\ \mathit{false}) \end{array}$$

  By induction, $\mathrm{read}(s',\ p \cdot \texttt{x} \cdot \texttt{:},\ \mathit{false}) \in$ *Stmt'*$_e$ so $t \in$ *Stmt'*$_{\texttt{x :}\ e}$.

The argument for the right-to-left direction is similar.

□

Lemma 5 (Parse Equivalence for StmtList)**.**
$\forall s, p \in$ *RegexPrefix*.

$$\begin{array}{rl} & s \in \mathit{StmtList}_e \\ \Leftrightarrow & \mathit{read}(s,\ p,\ \mathit{false}) \in \mathit{StmtList'}_e \end{array}$$

Proof. For the left-to-right direction we have two production rules for *StmtList*$_e$.

- $s \in \mathit{Stmt}_e$. This follows by Lemma 4 since $p \in$ *RegexPrefix*.
- $s \in \mathit{StmtList}_e\ \mathit{Stmt}_{e'}$. So $s = s' \cdot s''$ where $s' \in \mathit{StmtList}_e$ and $s'' \in \mathit{Stmt}_{e'}$. Then,

$$\begin{array}{rcl} t & = & \mathrm{read}(s' \cdot s'',\ p,\ \mathit{false}) \\ & = & \mathrm{read}(s',\ p,\ \mathit{false}) \cdot \mathrm{read}(s'',\ p \cdot \mathrm{read}(s',\ p,\ \mathit{false}),\ \mathit{false}) \end{array}$$

  By induction $\mathrm{read}(s',\ p,\ \mathit{false}) \in$ *StmtList'*$_e$ and by Lemma 4, $\mathrm{read}(s'',\ p \cdot \mathrm{read}(s',\ p,\ \mathit{false}),\ \mathit{false}) \in$ *Stmt'*$_{e'}$ (since by Lemma 13, $p \cdot \mathrm{read}(s',\ p,\ \mathit{false}) \in$ *RegexPrefix*). Thus, $t \in$ *StmtList'*$_{e\ e'}$.

The argument for the right-to-left direction is similar.

$\square$

Lemma 6 (Parse Equivalence for AssignExpr)**.**
$\forall s, p \in$ RegexPrefix. If $b = false$ then $s \neq$ `{` $\cdot\, s'$. If $b = true$ then s is unconstrained.

$$\begin{array}{rl} & s \in \mathit{AssignExpr}_e \\ \Leftrightarrow & \mathit{read}(s,\ p,\ b) \in \mathit{AssignExpr'}_e \end{array}$$

Proof. The constraint on s when b is false is due to the lookahead check in the production $\mathit{Stmt}_e ::= \mathit{AssignExpr}_e$ `;`. Lemma 11 will make use of this constraint.

For the left-to-right direction we have two production rules for $\mathit{AssignExpr}_e$.

- $s \in \mathit{BinaryExpr}_e$. This holds by Lemma 7.
- $s \in \mathit{CallExpr}_e$ `=` $\mathit{AssignExpr}_{e'}$. Then $s = s' \cdot$ `=` $\cdot\, s''$ where $s' \in \mathit{CallExpr}_e$ and $s'' \in \mathit{AssignExpr}_{e'}$. Then:

$$\begin{array}{rcl} t & = & \mathrm{read}(s' \cdot \texttt{=} \cdot s'',\ p,\ b) \\ & = & \mathrm{read}(s',\ p,\ b) \cdot \mathrm{read}(\texttt{=} \cdot s'',\ p \cdot \mathrm{read}(s',\ p,\ b),\ b) \\ & = & \mathrm{read}(s',\ p,\ b) \cdot \texttt{=} \cdot \mathrm{read}(s'',\ p \cdot \mathrm{read}(s',\ p,\ b) \cdot \texttt{=},\ b) \end{array}$$

  Since $p \in$ *RegexPrefix* by Lemma 8, $\mathrm{read}(s',\ p,\ b) \in \mathit{CallExpr'}_e$ and by induction $\mathrm{read}(s'',\ p{\cdot}\mathrm{read}(s',\ p,\ b){\cdot}\texttt{=},\ b) \in \mathit{AssignExpr'}_{e'}$ (since $p \cdot \mathrm{read}(s',\ p,\ b) \cdot \texttt{=} \in$ *RegexPrefix*) so $t \in \mathit{AssignExpr'}_{e\ \texttt{=}\ e'}$.

The argument for the right-to-left direction is similar.

$\square$

Lemma 7 (Parse Equivalence for BinaryExpr)**.**
$\forall s, p \in$ RegexPrefix. If $b = false$ then $s \neq$ `{` $\cdot\, s'$. If $b = true$ then s is unconstrained.

$$\begin{array}{rl} & s \in \mathit{BinaryExpr}_e \\ \Leftrightarrow & \mathit{read}(s,\ p,\ b) \in \mathit{BinaryExpr'}_e \end{array}$$

Proof. For the left-to-right direction there are three productions for $\mathit{BinaryExpr}_e$.

- $s \in \mathit{CallExpr}_e$. This holds by Lemma 8 since $p \in$ *RegexPrefix*.
- $s \in \mathit{BinaryExpr}_e$ `/` $\mathit{BinaryExpr}_{e'}$. We have $s = s' \cdot \texttt{/} \cdot s''$ where $s' \in \mathit{BinaryExpr}_e$ and $s'' \in \mathit{BinaryExpr}_{e'}$. Then:

$$\begin{array}{rcl} t & = & \mathrm{read}(s' \cdot \texttt{/} \cdot s'',\ p,\ b) \\ & = & \mathrm{read}(s',\ p,\ b) \cdot \mathrm{read}(\texttt{/} \cdot s'',\ p \cdot \mathrm{read}(s',\ p,\ b),\ b) \\ & = & \mathrm{read}(s',\ p,\ b) \cdot \texttt{/} \cdot \mathrm{read}(s'',\ p \cdot \mathrm{read}(s',\ p,\ b) \cdot \texttt{/},\ b) \\ & & (\text{since } p \cdot \mathrm{read}(s',\ p,\ b) \in \mathit{DividePrefix}) \end{array}$$

By induction $\text{read}(s',\ p,\ b) \in \mathit{BinaryExpr'}_{e}$ and $\text{read}(s'',\ p \cdot \text{read}(s',\ p,\ b) \cdot \texttt{/},\ b) \in \mathit{BinaryExpr'}_{e'}$ (since $p \cdot \text{read}(s',\ p,\ b) \cdot \texttt{/} \in \mathit{RegexPrefix}$) thus $t \in \mathit{BinaryExpr'}_{e\ \texttt{/}\ e'}$.

- $s \in \mathit{BinaryExpr}_{e}\ \texttt{+}\ \mathit{BinaryExpr}_{e'}$. We have $s = s' \cdot \texttt{+} \cdot s''$ where $s' \in \mathit{BinaryExpr}_{e}$ and $s'' \in \mathit{BinaryExpr}_{e'}$. Then:

$$\begin{array}{rcl} t & = & \text{read}(s' \cdot \texttt{+} \cdot s'',\ p,\ b) \\ & = & \text{read}(s',\ p,\ b) \cdot \texttt{+} \cdot \text{read}(s'',\ p \cdot \text{read}(s',\ p,\ b) \cdot \texttt{+},\ b) \end{array}$$

  By induction $\text{read}(s',\ p,\ b) \in \mathit{BinaryExpr'}_{e}$ and $\text{read}(s'',\ p \cdot \text{read}(s',\ p,\ b) \cdot \texttt{+},\ b) \in \mathit{BinaryExpr'}_{e'}$ (since $p \cdot \text{read}(s',\ p,\ b) \cdot \texttt{+} \in \mathit{RegexPrefix}$) thus $t \in \mathit{BinaryExpr'}_{e\ \texttt{+}\ e'}$.

The argument for the right-to-left direction is similar.

□

Lemma 8 (Parse Equivalence for CallExpr)**.**
$\forall s, p \in \mathit{RegexPrefix}$. If $b = \mathit{false}$ then $s \neq \texttt{\{} \cdot s'$. If $b = \mathit{true}$ then s is unconstrained.

$$\begin{array}{rl} & s \in \mathit{CallExpr}_{e} \\ \Leftrightarrow & \mathit{read}(s,\ p,\ b) \in \mathit{CallExpr'}_{e} \end{array}$$

Proof. For the left-to-right direction there are two production rules for $\mathit{CallExpr}_{e}$.

- $s \in \mathit{MemberExpr}_{e}\ \texttt{(}\mathit{AssignExpr'}_{e'}\texttt{)}$. We have $s = s' \cdot \texttt{(}s''\texttt{)}$ where $s' \in \mathit{MemberExpr}_{e}$ and $s'' \in \mathit{AssignExpr}_{e'}$. Then

$$\begin{array}{rcl} t & = & \text{read}(s' \cdot \texttt{(}s''\texttt{)},\ p,\ b) \\ & = & \text{read}(s',\ p,\ b) \cdot \text{read}(\texttt{(}s''\texttt{)},\ p \cdot \text{read}(s',\ p,\ b),\ b) \end{array}$$

  Since $p \in \mathit{RegexPrefix}$, by Lemma 9 we have $\text{read}(s',\ p,\ b) \in \mathit{MemberExpr'}_{e}$ and my Lemma 6 we have $\text{read}(s'',\ \epsilon,\ \mathit{false}) \in \mathit{AssignExpr'}_{e'}$ thus $t \in \mathit{CallExpr'}_{e\ \texttt{(}e'\texttt{)}}$.
- $s \in \mathit{CallExpr}_{e}\ \texttt{.}\ \texttt{x}$. Then $s = s' \cdot \texttt{.} \cdot \texttt{x}$ where $s' \in \mathit{CallExpr}_{e}$. Then

$$\begin{array}{rcl} t & = & \text{read}(s' \cdot \texttt{.} \cdot \texttt{x},\ p,\ b) \\ & = & \text{read}(s',\ p,\ b) \cdot \texttt{.} \cdot \texttt{x} \end{array}$$

  By induction $\text{read}(s',\ p,\ b) \in \mathit{CallExpr'}_{e}$. Thus $t \in \mathit{CallExpr'}_{e\ \texttt{.x}}$.

The argument for the right-to-left direction is similar.

□

Lemma 9 (Parse Equivalence for MemberExpr)**.**
$\forall s, p \in$ RegexPrefix. If $b = false$ then $s \neq$ `{` $\cdot\, s'$. If $b = true$ then s is unconstrained.

$$\begin{array}{rl} & s \in \textit{MemberExpr}_e \\ \Leftrightarrow & \textit{read}(s,\ p,\ b) \in \textit{MemberExpr'}_e \end{array}$$

Proof. For the left-to-right direction there are two three production rules for *MemberExpr*$_e$.

- $s \in$ *PrimaryExpr*$_e$. This follows from Lemma 11 since $p \in$ *RegexPrefix*.
- $s \in$ *FunctionExpr*$_e$. This follows from Lemma 10.
- $s \in$ *MemberExpr*$_e$ `. x`. We have $s = s' \cdot$ `.x` where $s \in$ *MemberExpr*$_e$. Then

$$\begin{array}{rcl} t & = & \text{read}(s' \cdot \texttt{.} \cdot \texttt{x},\ p,\ b) \\ & = & \text{read}(s',\ p,\ b) \cdot \texttt{.} \cdot \texttt{x} \end{array}$$

  By induction read$(s',\ p,\ b) \in$ *MemberExpr'*$_e$ thus $t \in$ *MemberExpr'*$_{e\texttt{.x}}$.

The argument for the right-to-left direction is similar.

□

Lemma 10 (Parse Equivalence for FunctionExpr)**.**
$\forall s, b, p \in$ RegexPrefix.

$$\begin{array}{rl} & s \in \textit{FunctionExpr}_e \\ \Leftrightarrow & \textit{read}(s,\ p,\ b) \in \textit{FunctionExpr'}_e \end{array}$$

Proof. Since $s =$ `function` x `(`x'`)` `{`s'`}` where $s' \in$ *SourceElements*$_e$ and

$$t = \text{read}(s,\ p,\ b) = \texttt{function} \cdot x \cdot \texttt{(}x'\texttt{)} \cdot \texttt{\{}t'\texttt{\}}$$

where $t' = \text{read}(s',\ \epsilon,\ \textit{false})$ and by Lemma 1, $t' \in$ *SourceElements'*$_e$ we have $t \in$ *FunctionExpr'*$_{\texttt{function}\ x\ \texttt{(}x\texttt{)}\ \texttt{\{}e\texttt{\}}}$.
The argument for the right-to-left direction is similar.

□

Lemma 11 (Parse Equivalence for PrimaryExpr)**.**
$\forall s, p \in$ RegexPrefix. If $b = false$ then $s \neq$ `{` $\cdot\, s'$. If $b = true$ then s is unconstrained.

$$\begin{array}{rl} & s \in \textit{PrimaryExpr}_e \\ \Leftrightarrow & \textit{read}(s,\ p,\ b) \in \textit{PrimaryExpr'}_e \end{array}$$

Proof. For the left-to-right direction there are several production rules for *PrimaryExpr*$_e$.

- $s \in$ `x`. Then $s =$ `x` and read$(\texttt{x},\ p,\ b) \in$ *PrimaryExpr'*$_{\texttt{x}}$ directly.

- $s \in$ `/`$\cdot$`x`$\cdot$`/`. Then $s =$ `/`$\cdot$`x`$\cdot$`/` and read(`/`$\cdot$`x`$\cdot$`/`, p, b) $=$ `/`x`/` $\in$ *PrimaryExpr'*$_{/x/}$ since $p \in$ *RegexPrefix*.
- $s \in$ `{x:`*AssignExpr*$_e$`}`. Then $s =$ `{`$\cdot$`x`$\cdot$`:`$\cdot s' \cdot$`}` where $s' \in$ *AssignExpr*$_e$. Then:

$$\begin{array}{rcl} t & = & \text{read}(\texttt{\{} \cdot \texttt{x} \cdot \texttt{:} \cdot s' \cdot \texttt{\}},\ p,\ \mathit{true}) \\ & = & \texttt{\{x} \cdot \texttt{:} \cdot t'\texttt{\}} \end{array}$$

  where $t' = \text{read}(s',\ \texttt{x} \cdot \texttt{:},\ \mathit{true})$. By Lemma 6, $t' \in$ *AssignExpr'*$_e$ and thus $t \in$ *PrimaryExpr'*$_{\texttt{\{x:}e\texttt{\}}}$.
- $s \in$ `(`*AssignExpr*$_e$`)`. Then $s =$ `(`s'`)` where $s' \in$ *AssignExpr*$_e$. So,

$$\begin{array}{rcl} t & = & \text{read}(\texttt{(}s'\texttt{)},\ p,\ b) \\ & = & \texttt{(}t'\texttt{)} \end{array}$$

  where $t' = \text{read}(s',\ \epsilon,\ \mathit{false})$. By Lemma 6 $t' \in$ *AssignExpr'*$_e$. So, $t \in$ *PrimaryExpr'*$_{\texttt{(}e\texttt{)}}$.

For the right-to-left direction the argument is similar.

□

Lemma 12 (SourceElement Prefix)**.**
$\forall s, p \in$ *RegexPrefix*.

$$\begin{array}{rl} & s \in \mathit{SourceElement}_e \\ \Rightarrow & \mathit{read}(s,\ p,\ \mathit{false}) \in \mathit{RegexPrefix} \end{array}$$

Proof. There are two cases:

- $s \in$ *Stmt*$_e$. This holds by Lemma 13.
- $s \in$ *FunctionDecl*$_e$. Follows directly.

□

Lemma 13 (Stmt Prefix)**.**
$\forall s, p \in$ *RegexPrefix*.

$$\begin{array}{rl} & \mathit{read}(s,\ p,\ \mathit{false}) \in \mathit{Stmt'}_e \\ \Rightarrow & p \cdot \mathit{read}(s,\ p,\ \mathit{false}) \in \mathit{RegexPrefix} \end{array}$$

Proof. We have several cases:

- read(s, p, *false*) $\in$ `{`*StmtList'*$_e$`}`. Follows since $p \in$ *RegexPrefix*.
- read(s, p, *false*) $\in$ *AssignExpr'*$_e$ `;`. Follows since $t \cdot$ `;` $\in$ *RegexPrefix* for any t.
- $t =$ read(s, p, *false*) $\in$ `if (`*AssignExpr'*$_e$`)` *Stmt*$_{e'}$. Since $t =$ `if` $\cdot$ `(`t'`)` $\cdot\, t''$ where $t'' \in$ *Stmt'*$_{e'}$ by induction $t'' \in$ *RegexPrefix* and thus $p \cdot t \in$ *RegexPrefix*.
- read(s, p, *false*) $\in$ `return`. Follows since $p \cdot$ `return` $\in$ *RegexPrefix*.
- $t =$ read(s, p, *false*) $\in$ `return` *AssignExpr'*$_e$ `;`. Since `;` $\in$ *RegexPrefix* then $p \cdot t \in$ *RegexPrefix*.

- $t = \text{read}(s,\ p,\ \mathit{false}) \in \texttt{x} : \mathit{Stmt'}_e$. Since $t = \texttt{x} \cdot : \cdot t'$ where $t' \in \mathit{Stmt'}_e$. Since $p \in$ *RegexPrefix* and by induction $t' \in$ *RegexPrefix* we have $p \cdot t \in$ *RegexPrefix*.

$\square$