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 [3, 6] 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:

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

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 [1].

Recently the Honu project [7, 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. 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 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` $\cdot$ `(` $\cdot$ `/` $\cdot$ `x` $\cdot$ `/` $\cdot$ `)`.

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

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

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.

Other cases require additional context to disambiguate. For example, if the previous token tree was a parentheses (e.g. dividing the result of a call `foo(100)/ 10`) then slash will be the divide operator *unless* the token tree before the parentheses was the keyword `if` in which case 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 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.

3.1 Proving Read

To show that our read algorithm correctly distinguishes between the divide operator and a regular expression literal,

Figure 1: AST for Simplified JavaScript

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

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 [5] 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:

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

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 $t = \text{read}(s,\ \epsilon,\ \mathit{false})$, and then parses this token tree stream t into an AST e.

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

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

Figure 2: Simplified Read Algorithm

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

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

$$
\begin{array}{lcl}
\text{read} : \textit{Token}^* \to \textit{TokenTree}^* \to \textit{Bool} \to \textit{TokenTree}^* & & \\
\text{read}(\texttt{/} \cdot x \cdot \texttt{/} \cdot s,\ \epsilon,\ b) & = & \texttt{/}x\texttt{/} \cdot \text{read}(s,\ \texttt{/}x\texttt{/},\ b) \\
\text{read}(\texttt{/} \cdot x \cdot \texttt{/} \cdot s,\ p \cdot p',\ b) & = & \texttt{/}x\texttt{/} \cdot \text{read}(s,\ p \cdot p' \cdot \texttt{/}x\texttt{/},\ b) \\
\quad \textit{if } p' \in \textit{Punctuator or Keyword} & & \\
\text{read}(\texttt{/} \cdot x \cdot \texttt{/} \cdot s,\ p \cdot \texttt{if} \cdot \texttt{(}t\texttt{)},\ b) & = & \texttt{/}x\texttt{/} \cdot \text{read}(s,\ p \cdot \texttt{if} \cdot \texttt{(}t\texttt{)} \cdot \texttt{/}x\texttt{/},\ b) \\
\text{read}(\texttt{/} \cdot x \cdot \texttt{/} \cdot s,\ p \cdot p' \cdot \texttt{function}^l \cdot x \cdot \texttt{(}t\texttt{)} \cdot \texttt{\{}t'\texttt{\}},\ b) & = & \texttt{/}x\texttt{/} \cdot \text{read}(s,\ p \cdot p' \cdot \texttt{function}^l \cdot x \cdot \texttt{(}t\texttt{)} \cdot \texttt{\{}t'\texttt{\}} \cdot \texttt{/}x\texttt{/},\ b) \\
\quad \textit{if false} = \text{isExprPrefix}(p',\ b,\ l) & & \\
\text{read}(\texttt{/} \cdot x \cdot \texttt{/} \cdot s,\ p \cdot \texttt{\{}t\texttt{\}}^l,\ b) & = & \texttt{/}x\texttt{/} \cdot \text{read}(s,\ p \cdot \texttt{\{}t\texttt{\}}^l \cdot \texttt{/}x\texttt{/},\ b) \\
\quad \textit{where false} = \text{isExprPrefix}(p,\ b,\ l) & &
\end{array}
$$

$$
\begin{array}{lcl}
\text{read}(\texttt{/} \cdot s,\ p \cdot x,\ b) & = & \texttt{/} \cdot \text{read}(s,\ p \cdot x \cdot \texttt{/},\ b) \\
\text{read}(\texttt{/} \cdot s,\ p \cdot \texttt{/}x\texttt{/},\ b) & = & \texttt{/} \cdot \text{read}(s,\ p \cdot \texttt{/}x\texttt{/} \cdot \texttt{/},\ b) \\
\text{read}(\texttt{/} \cdot s,\ p \cdot \texttt{(}t\texttt{)},\ b) & = & \texttt{/} \cdot \text{read}(s,\ p \cdot \texttt{(}t\texttt{)} \cdot \texttt{/},\ b) \\
\text{read}(\texttt{/} \cdot s,\ p \cdot p' \cdot \texttt{function}^l \cdot x \cdot \texttt{(}t\texttt{)} \cdot \texttt{\{}t'\texttt{\}},\ b) & = & \texttt{/} \cdot \text{read}(s,\ p \cdot p' \cdot \texttt{function}^l \cdot x \cdot \texttt{(}t\texttt{)} \cdot \texttt{\{}t'\texttt{\}} \cdot \texttt{/},\ b) \\
\quad \textit{if true} = \text{isExprPrefix}(p',\ b,\ l) & & \\
\text{read}(\texttt{/} \cdot s,\ p \cdot \texttt{\{}t\texttt{\}}^l,\ b) & = & \texttt{/} \cdot \text{read}(s,\ p \cdot \texttt{\{}t\texttt{\}}^l \cdot \texttt{/},\ b) \\
\quad \textit{where true} = \text{isExprPrefix}(p,\ b,\ l) & &
\end{array}
$$

$$
\begin{array}{lcl}
\text{read}(\texttt{(} \cdot s \cdot \texttt{)} \cdot s',\ p,\ b) & = & \texttt{(}t\texttt{)} \cdot \text{read}(s',\ p \cdot \texttt{(}t\texttt{)},\ b) \\
\quad \textit{where } s \textit{ contains no unmatched } \texttt{)} & & \quad \textit{where } t = \text{read}(s,\ \epsilon,\ \textit{false}) \\
\text{read}(\texttt{\{}^l \cdot s \cdot \texttt{\}} \cdot s',\ p,\ b) & = & \texttt{\{}t\texttt{\}}^l \cdot \text{read}(s',\ p \cdot \texttt{\{}t\texttt{\}}^l,\ b) \\
\quad \textit{where } s \textit{ contains no unmatched } \texttt{\}} & & \quad \textit{where } t = \text{read}(s,\ \epsilon,\ \text{isExprPrefix}(p,\ b,\ l)) \\
\text{read}(x \cdot s,\ p,\ b) & = & x \cdot \text{read}(s,\ p \cdot x,\ b) \\
\text{read}(\epsilon,\ p,\ b) & = & \epsilon
\end{array}
$$

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 syntax 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 *enforestation* [7]. Enforestation works by progressively recognizing syntax forms (e.g. literals, identifiers, expressions, and statements) during expansion.

Figure 3: Simplified ES5 Grammar

$$
\begin{array}{lcl}
\mathit{PrimaryExpr}_{x} & ::= & x \\
\mathit{PrimaryExpr}_{/x/} & ::= & \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{FunctionExpr}_{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{FunctionDecl}_{\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{FunctionExpr}_{\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{FunctionDecl}_{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}
$$

A term tree is a kind of proto-AST that represents a partial parse of the program. As the expander passes through the token tress, 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 a term tree). 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: {
   <id: 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 as binary expressions such as `<op: +, left: 40, right: 2>`.

The enforestation technique described here was first proposed for the Honu language [7]. We take their technique and extend it with two features described in the following sections: infix macros and the invoke pattern class.

4.1 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.

For example, the upcoming ES6 version of JavaScript includes a shorthand syntax for defining functions with arrow notation:

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

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.

Honu addressed this need for more flexible syntax transformation forms in a limited fashion via user definable unary and binary operators[2]. Definable operators provide the ability to set custom precedence and associativity along with a syntax transformation. For example, the exponentiation operator could be defined as:

```
operator ^ 10 left {$lhs, $rhs} => {
  Math.pow($lhs, $rhs)
}
2 ^ 100;
// expands to:
// Math.pow(2, 100);
```

Unfortunately, definable operators are limited in that their operands must be expressions (this limitation allows setting custom precedence and associativity rules). This means it is impossible to define syntax forms such as arrow functions via definable operators since the parameter list is not an expression.

To address this limitation, sweet.js introduces *infix macros*, which allow a macro to match syntax that comes before 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 the cursor in the token stream).

For example, the following infix macro implements arrow functions:

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

var id = (x) => x
```

[2] Sweet.js will also provide support for definable operators though at the time of this writing they have not yet been fully implemented.

Standard 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:

$$\mathit{transformer} : \mathit{TokenTree}^* \rightarrow \mathit{TokenTree}^*$$

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 ends as needed, yielding new preceding and following tokens:

$$\begin{array}{rcl} \mathit{transfomer}_{\mathit{infix}} & : & (\mathit{TokenTree}^*, \mathit{TokenTree}^*) \\ & \rightarrow & (\mathit{TokenTree}^*, \mathit{TokenTree}^*) \end{array}$$

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.

$$\begin{array}{rcl} \mathit{transfomer}_{\mathit{infix}} & : & ((\mathit{TokenTree}^*, \mathit{TermTree}^*), \mathit{TokenTree}^*) \\ & \rightarrow & (\mathit{TokenTree}^*, \mathit{TokenTree}^*) \end{array}$$

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 definable 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 was with operators. It is unclear if this is a fundamental limitation or if there is some technique that might enable setting 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:

```
macro id { rule { $x } => { $x }}
macro m {
  rule infix { $first $second | } => {
    $first + $second
  }
}
foo("sweet")
id 100 m
// expands to:
// foo
// ("sweet") + 100
```

This behavior introduces an asymmetry in 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 are a nice complement to definable operators and greatly extend the kinds of syntax forms that can be written as a macro.

4.2 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 (,) ...];
  }
}
```

...?

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 [2, 4] of tracking the lexical context in each syntax object.

todo: do we need examples of hygiene?

5.1 Macro Binding Limitation

Unfortunately, the syntax of JavaScript does place a limitations on our system that is not present in Scheme. In Scheme, definitions and uses of macros can 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. First to register any macro definitions and second to expand the macros that were 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;
}
// fails!
```

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.

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. 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 [6].

- Scheme/Racket
- Honu

- Template Haskell
- Nemerle [9]
- Scala
- Closure

8. Conclusion

References

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

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

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

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

[5] 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.

[6] 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.

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

[8] 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.

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

A. Read Proof

todo: define RegexPrefix/DivPrefix

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 $s'' \in \mathit{SourceElement}_{e'}$.

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

By induction $\text{read}(s',\ \epsilon,\ \mathit{false}) \in \mathit{SourceElements'}_e$ and by Lemma 2, $\text{read}(s'',\ \text{read}(s',\ \epsilon,\ \mathit{false}),\ \mathit{false}) \in \mathit{SourceElement'}_{e'}$ (since by Lemma 12, $\text{read}(s',\ \epsilon,\ \mathit{false}) \in \mathit{RegexPrefix}$). 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)**.**
$\forall s, p, b.$

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

Proof. For the left-to-right direction there is one production rule for $\mathit{FunctionDecl}_e$:

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

We have $s = \texttt{function}\ x\ \texttt{(}x\texttt{)}\ \texttt{\{}s'\texttt{\}}$ where $s' \in \mathit{SourceElements}_e$ 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 $t' = \text{read}(s',\ \epsilon,\ \mathit{false})$. Since by Lemma 1, $t' \in \mathit{SourceElements'}_e$ we have $t \in \mathit{FunctionDecl'}_{\texttt{function}\ x\ \texttt{(}x\texttt{)}\ \texttt{\{}e\texttt{\}}}$.

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

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

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

Proof. For the left-to-right direction we have several production rules for Stmt_e.

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

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

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

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

Since $p \in \mathit{RegexPrefix}$ by Lemma 6, $\text{read}(s',\ p,\ \mathit{false}) \in \mathit{AssignExpr'}_e$ we have $t \in \mathit{Stmt'}_e$.
- $s \in \texttt{if (}\mathit{AssignExpr}_e\texttt{)}\ \mathit{Stmt}_{e'}$. We have $s = \texttt{if} \cdot \texttt{(}s'\texttt{)} \cdot s''$ where $s' \in \mathit{AssignExpr}_e$ and $s'' \in \mathit{Stmt}_{e'}$.

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

where

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

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

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

By Lemma 6, $\text{read}(s',\ p \cdot \texttt{return},\ \mathit{false}) \in \mathit{AssignExpr'}_e$ thus $t \in \mathit{Stmt'}_{\texttt{return}\ e}$.

- $s \in \texttt{x} : \mathit{Stmt}_e$. We have $s = \texttt{x} \cdot : \cdot s'$ where $s' \in \mathit{Stmt}_e$. Then:

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

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

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

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

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

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

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

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

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

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

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

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

Proof. The constraint on s when b is false is due to the lookahead check in the production $\mathit{Stmt}_e ::= \mathit{AssignExpr}_e\ \texttt{;}$. 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.
- $s \in \mathit{CallExpr}_e\ \texttt{=}\ \mathit{AssignExpr}_{e'}$. Then $s = s' \cdot \texttt{=} \cdot s''$ where $s' \in \mathit{CallExpr}_e$ and $s'' \in \mathit{AssignExpr}_{e'}$. Then:

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

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

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

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

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

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

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

  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 $t \in \mathit{CallExpr'}_{e\ \texttt{(}e'\texttt{)}}$.

- $s \in$ *CallExpr*$_e$ `.x`. Then $s = s' \cdot \texttt{.} \cdot \texttt{x}$ where $s' \in$ *CallExpr*$_e$. Then

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

  By induction read$(s',\ p,\ b) \in$ *CallExpr'*$_e$. Thus $t \in$ *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* $s \neq \texttt{\{} \cdot s'$. *If* $b =$ *true then* s *is unconstrained.*

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

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 $s = s' \cdot \texttt{.x}$ where $s \in$ *MemberExpr*$_e$. Then

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

  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*.

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

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 $t' =$ read$(s',\ \epsilon,$ *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* $s \neq \texttt{\{} \cdot s'$. *If* $b =$ *true then* s *is unconstrained.*

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

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

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

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

  where $t' =$ read$(s',$ `x` $\cdot$ `:`$,$ *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,

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

  where $t' =$ read$(s',\ \epsilon,$ *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*.

$$\begin{aligned} & s \in \textit{SourceElement}_e \\ \Rightarrow\ & \textit{read}(s,\ p,\ \textit{false}) \in \textit{RegexPrefix} \end{aligned}$$

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*.

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

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

□