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. It's pretty sweet.

1. Introduction

Expressive macros systems have a long history in the design of extensible programming languages going back to Lisp and Scheme [9, 14] as a powerful tool that enables programmers to craft their own languages.

While macro systems have found success in Lisp-derived languages, they have not been widely adopted in languages such as JavaScript. In part, this failure is due to the difficulty in implementing macro systems in languages that are not fully delimited. A key feature of a sufficiently expressive macro system is the ability for macros to manipulate unparsed and unexpanded subexpressions. In a language with parentheses like Scheme, manipulating unparsed subexpressions is simple:

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

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

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

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

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

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

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

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

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

Once JavaScript source has been correctly read, there are still a number of challenges to building an expressive macro system. The lack of parentheses in particular make writing

declarative macro definitions difficult. For example, the `if` statement in JavaScript allows undelimited then and else branches:

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

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

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

This approach, called *enforestation*, was pioneered by Honu [18, 19], which we adapt here for JavaScript[1]. In addition, we make two extensions to the Honu technique that enable more expressive macros to be built. First, as described in Section 4.1 we add support for infix macros, which allow macros to match syntax both before and after the macro identifier. Secondly, we implement the `invoke` pattern class, described in Section 4.2, which allows macro authors to extend the patterns used to match syntax.

Sweet.js is implemented in JavaScript and takes source code written with sweet.js macros and outputs the expanded source that can be run in any JavaScript environment. The project web page[2] includes an interactive editor that makes it simple to try out writing macros without requiring any installation. There is already an active community using sweet.js to implement significant features from the upcoming ES6 version of JavaScript [15] and pattern matching for JavaScript [6].

We make the following contributions:

- a reader implementation that addresses the complexities of JavaScript, including regular expression literals, together with a correctness proof for this reader.
- a macro system with 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. Reading JavaScript

Parsers give structure to unstructured source code. In parsers without macro systems this is usually accomplished by a lexer (which converts a character stream to a token stream) and a parser (which converts the token stream into an AST according to a context-free grammar). A system with macros must implement a macro *expander* that sits between the lexer and parser. Some macros systems, such as the C preprocessor [11], work over just the token stream. However, to implement truly expressive Scheme-like macros that can manipulate groups of unparsed tokens, it is necessary to structure the token stream via a *reader*, which performs delimiter matching and enables macros to manipulate delimiter grouped tokens.

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

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

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

In our formalism in Figure 2, read takes a *Token* sequence. Tokens are the output of a very simple lexer, which we do not define here, that preforms the standard grouping of keywords, puncutators, the (unmatched) delimiters, and variable identifiers but does not distinguish between regular expressions and divide (i.e. there is just the ambiguous `/` token).

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

The job of read is then to produces a correct *TokenTree* sequence. Token trees include regular expression literals $\texttt{/}x\texttt{/}$ (where x is the regular expression body, which we simplify here to a variable rather than include irrelevant details of parsing a regular expression body) and fully matched delimiters with nested token tree sequences $\underline{\texttt{(}t\texttt{)}}$ and $\underline{\texttt{\{}t\texttt{\}}}$ rather than individual delimiters (note that we write token tree delimiters with an underline to distinguish them from token delimiters).

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

[1] Honu does not support regular expression literals which simplifies their reader.

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

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

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

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

However, this next function returns `undefined` because the grammar calls for a semicolon to be inserted after the `return` keyword.

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

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

We write a sequence by separating elements with a dot so the source string "`foo(/x/)`" is lexed into a sequence of six tokens `foo` · `(` · `/` · `x` · `/` · `)`. The equivalent token tree sequence is `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. Not that this may correspond to looking back an unbounded distance in the original token stream.

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

Other cases require additional context to disambiguate. For example, if the previous token tree was a parentheses (e.g. dividing the result of a call `foo` · `(` · `x` · `)` · `/` · `y`) 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/   // regex
```

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/  // regex
}
```

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

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

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

The reader handles these cases by checking if the prefix of a curly brace 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.

The type for read is then:

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

Read is a function that takes a sequence of tokens, a prefix of previously read token trees, a boolean indicating if the token stream currently being read is inside an object literal, and returns sequence of token trees.

The implementation of read shown in Figure 2 includes an auxiliary function isExprPrefix used to determine if the prefix for a curly brace indicates that the braces should be part of an expression (i.e. the braces are an object literal) or if they should be a block statement.

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

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

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

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

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

2.1 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

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

we show that a parser defined over normal tokens produces the same AST as a parser defined over the token trees produced from read.

The parser for normal tokens is defined in Figure 3, and generates ASTs in the abstract syntax shown in Figure 1. 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 5.1 standard [13] and many of the nonterminal names we use here correspond to shortened versions of nonterminals in the standard.

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:

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

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. □

3. Writing Macros

The sweet.js system provides two kinds of macros: *rule* macros (analogous to `syntax-rules` in Scheme [5]) and *case* macros (analogous to `syntax-case` in Scheme [12]). Rule macros are the simpler of the two and work by matching on a pattern and generating a template:

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

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

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

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

The above example 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 the largest possible expression, so in this example the entire expression `40 + 2` is bound to `$init`.

Along with the template-based rule macros, sweet.js also provides the more powerful procedural *case* macros. Instead of a template, the body of a case macro is JavaScript code that runs when the macro is invoked. Rather than be the entire body of a macro, templates are interspersed with JavaScript code by using the notation `#{ ... }`.

For example, the following macro creates a string from the contents of a file.

```
macro fromFile {
```

Figure 2: Simplified Read Algorithm

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

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

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

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

Here we take the file name token matched by the macro, unwrap it, and read[4] the file into a variable. The `makeValue` function is used to create a string syntax object (`#{here}` is just a placeholder for the lexical context) from the context of the file and `letstx` (analogous to `with-syntax` in Scheme) binds that new syntax object to a pattern variable `$content` that can be used inside of a template.

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

Figure 3: Simplified ES5 Grammar

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

4. Enforestation

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

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

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

Sweet.js groups tokens by transforming a token tree into a *term tree* through *enforestation* [18]. Enforestation works by progressively recognizing syntax forms (e.g. literals, identifiers, expressions, and statements) during expansion.

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

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

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

Enforestation begins by loading the `let` macro into the macro environment and converting the function declaration into a term tree (we use angle brackets to denote 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: {
   <var: x, init: <op: +, left: 40, right: 2>
   <return: <op: +, left: x right: y>
 }>
<call: foo, args: (100)>
```

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

The enforestation technique described here was first proposed for the Honu language [18]. 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[5]. 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);
```

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

Unfortunately, definable operators are limited in that their operands must be expressions (this limitation allows setting custom precedence and associativity rules). This restriction 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
// expands to:
// var id = function (x) { return x; }
```

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

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

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

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

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

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

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

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

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

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

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

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

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

is first enforested to:

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

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

While infix macros fill the gap left by 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 as with operators. It is unclear if this is a fundamental limitation or if there is some technique that might allow custom precedence and associativity for infix macros. The second limitation is that the preceding tokens to an infix macro must be first expanded. This means that any macros that occur before an infix macro will be invoked first:

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

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

4.2 Invoke and Pattern Classes

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

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

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

```
// r = ["#FF0000"]
// g = ["#00FF00"]
```

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

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

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

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

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

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

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

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

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

5. Hygiene

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

Our implementation of hygiene for sweet.js follows the Scheme approach [7, 12] of tracking the lexical context in each syntax object.

5.1 Macro Binding Limitation

Unfortunately, the syntax of JavaScript does place a limitations on our system that is not present in Scheme. 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. The first pass registers any macro definitions and the second pass expands any uses of macros discovered during the first pass. Critically, during the first pass the expander does not expand inside internal definitions.

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

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

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

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

Scheme is able to defer expansion of macros inside of internal definitions because internal definitions are fully delimited. The expander can skip over all of the syntax inside of the delimiter because there actually is an *inside* to skip over. However, this is not true for JavaScript since `var` statements

(JavaScript's equivalent of Scheme's internal definitions) are not delimited and so it is not possible to use a macro that has not yet been defined in a `var` statement in sweet.js.

For example, this will fail:

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

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

Macros have been extensively used and studied in Lisp [9, 17] and related languages for many years. Scheme in particular has embraced macros, pioneering the development of declarative definitions [14] and working out the hygiene conditions for term rewriting macros (rule macros) [5] and procedural macros (case macros) [12] that enable true composability. In addition there has been work to integrate procedural macros and module systems [8, 10].

Racket takes it even further by extending the Scheme macro system with deep hooks into the compilation process [7, 23].

In addition, there are a number of macro systems for languages with more traditional syntax that are not fully delimited.

As mentioned before, Honu [18, 19] is the primary inspiration for sweet.js. In contrast with Honu, which does not include regular expression literals, we solve the reader problem for JavaScript and introduce infix macros along with the `invoke` pattern class.

Macro systems that use a similar technique as Honu include Fortress [2] and Dylan [3] however they only provide support for term rewriting macros (our `rule` macros). Dylan's auxilary rules are similar to our `invoke` pattern class. Nemerle [21] also uses a similar technique but does not allow local definitions of macros.

Template Haskell [20] does AST matching and makes a treadoff by forcing the macro call site to always be demarcated.

Scala macros [4].

MetaCaml [16, 22].

C++ templates [1].

References

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

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

[3] Jonathan Bachrach, Keith Playford, and Chandler Street. D-Expressions : Lisp Power , Dylan Style. *Style DeKalb IL*, 1999.

[4] Eugene Burmako. Scala macros: let our powers combine! In *Proceedings of the 4th Workshop on Scala - SCALA '13*, pages 1–10, New York, New York, USA, July 2013. ACM Press.

[5] 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.

[6] Nate Faubion. The Sparkler project. `https://github.com/natefaubion/sparkler`.

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

[8] Matthew Flatt. Composable and compilable macros: You Want it When? *ACM SIGPLAN Notices*, 37(9):72–83, September 2002.

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

[10] Abdulaziz Ghuloum and R. Kent Dybvig. Implicit phasing for R6RS libraries. *ACM SIGPLAN Notices*, 42(9):303, October 2007.

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

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

[13] 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.

[14] 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.

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

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

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

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

[19] 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.

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

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

[22] W Taha and T Sheard. Multi-stage programming with explicit annotations. *ACM SIGPLAN Notices*, 1997.

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

A. Read Proof

To help reason about prefixes, we use two disjoint sets, *RegexPrefix* and *DividePrefix*, that contain the prefixes that determine if a slash should be ether a divide or the start of a regular expression. Theses sets are parametrized by a boolean indicating if the prefix is inside an object literal or a block statement:

$$
\begin{array}{rcl}
\mathit{RegexPrefix}_b & ::= & \epsilon \\
& | & p \cdot k \quad \textit{if } k \in \mathit{Punctuator} \cup \mathit{Keyword} \\
& | & p \cdot \texttt{if} \cdot \underline{(t)} \\
& | & p \cdot \texttt{function}^l \cdot x \cdot \underline{(t)} \cdot \underline{\{t'\}} \\
& & \quad \textit{if } \text{isExprPrefix}(p,\ b,\ l) = \mathit{false} \\
& | & p \cdot \underline{\{t\}}^l \\
& & \quad \textit{if } \text{isExprPrefix}(p,\ b,\ l) = \mathit{false}
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{DividePrefix}_b & ::= & p \cdot x \\
& | & p \cdot /x/ \\
& | & p \cdot k \cdot \underline{(t)} \quad \textit{if } k \neq \texttt{if} \\
& & \quad \textit{if } \text{isExprPrefix}(p,\ b,\ l) = \mathit{true} \\
& | & p \cdot \underline{\{t\}}^l \\
& & \quad \textit{if } \text{isExprPrefix}(p,\ b,\ l) = \mathit{true}
\end{array}
$$

These sets are the prefixes used by the read function in Figure 2.

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.
- $s \in \mathit{SourceElements}_e \cdot \mathit{SourceElement}_{e'}$. 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 11, $\text{read}(s',\ \epsilon,\ \mathit{false}) \in \mathit{RegexPrefix}_b$). Thus $t \in \mathit{SourceElements'}_{e\ e'}$.

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

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

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

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

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

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

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

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

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

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

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

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)**.**
$\forall s, p \in \mathit{RegexPrefix}_b.$

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

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

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

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

where $t' = \text{read}(s',\ \epsilon,\ \mathit{false})$. By Lemma 5, $t' \in \mathit{StmtList'}_e$ so $t \in \mathit{Stmt'}_{\underline{\{e\}}}$.

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

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

  where

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

  By Lemma 6 $t' \in \mathit{AssignExpr'}_e$. By induction, $t'' \in \mathit{Stmt'}_{e'}$ (since $p \cdot \underline{\texttt{(}\ t'\ \texttt{)}} \in \mathit{RegexPrefix}_b$). 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.
- $s \in \texttt{return} \cdot \mathit{AssignExpr}_e \texttt{;}$. 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 $t \in \mathit{Stmt'}_{\texttt{return}\ e}$.
- $s \in \texttt{x} \cdot \texttt{:} \cdot \mathit{Stmt}_e$. We have $s = \texttt{x} \cdot \texttt{:} \cdot s'$ where $s' \in \mathit{Stmt}_e$. Then:

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

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

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

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

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

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 $p \in \mathit{RegexPrefix}_b$.
- $s \in \mathit{StmtList}_e \cdot \mathit{Stmt}_{e'}$. 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 12, $p \cdot \text{read}(s',\ p,\ \mathit{false}) \in \mathit{RegexPrefix}_b$). Thus, $t \in \mathit{StmtList'}_{e\ e'}$.

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

Lemma 6 (Parse Equivalence for AssignExpr)**.**
$\forall s, p \in \mathit{RegexPrefix}_b$. *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 10 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.
- $s \in \mathit{CallExpr}_e \cdot \texttt{=} \cdot \mathit{AssignExpr}_{e'}$. 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 $p \in \mathit{RegexPrefix}_b$ 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 $p \cdot \text{read}(s',\ p,\ b) \cdot \texttt{=} \in \mathit{RegexPrefix}_b$) so $t \in \mathit{AssignExpr'}_{e\ \texttt{=}\ e'}$.

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

Lemma 7 (Parse Equivalence for BinaryExpr)**.**
$\forall s, p \in \mathit{RegexPrefix}_b$. *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 $p \in \mathit{RegexPrefix}_b$.
- $s \in \mathit{BinaryExpr}_e \cdot \texttt{/} \cdot \mathit{BinaryExpr}_{e'}$. We have $s = s' \cdot \texttt{/} \cdot s''$ where $s' \in \mathit{BinaryExpr}_e$ and $s'' \in \mathit{BinaryExpr}_{e'}$. Then:

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

  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 $p \cdot \text{read}(s',\ p,\ b) \cdot \texttt{/} \in \mathit{RegexPrefix}_b$) thus $t \in \mathit{BinaryExpr'}_{e\ \texttt{/}\ e'}$.
- $s \in \mathit{BinaryExpr}_e \cdot \texttt{+} \cdot \mathit{BinaryExpr}_{e'}$. 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 read$(s',\ p,\ b) \in$ *BinaryExpr'*$_e$ and read$(s'',\ p \cdot \text{read}(s',\ p,\ b) \cdot \texttt{+},\ b) \in$ *BinaryExpr'*$_{e'}$ (since $p \cdot \text{read}(s',\ p,\ b) \cdot \texttt{+} \in$ *RegexPrefix*$_b$) thus $t \in$ *BinaryExpr'*$_{e\ \texttt{+}\ e'}$.

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

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

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

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

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

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

  Since $p \in$ *RegexPrefix*$_b$, by Lemma 9 we have read$(s',\ p,\ b) \in$ *MemberExpr'*$_e$ and my Lemma 6 we have read$(s'',\ \epsilon,$ *false*$) \in$ *AssignExpr'*$_{e'}$ thus $t \in$ *CallExpr'*$_{e\ \texttt{(}e'\texttt{)}}$.
- $s \in$ *CallExpr*$_e \cdot \texttt{.} \cdot \texttt{x}$. Then $s = s' \cdot \texttt{.} \cdot \texttt{x}$ where $s' \in$ *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 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$_b$. If $b =$ false then $s \neq \texttt{\{} \cdot s'$. 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 10 since $p \in$ *RegexPrefix*$_b$.
- $s \in$ *Function*$_e$. This follows from Lemma 3.
- $s \in$ *MemberExpr*$_e \cdot \texttt{.} \cdot \texttt{x}$. We have $s = s' \cdot \texttt{.x}$ 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 PrimaryExpr)**.**
$\forall s, p \in$ RegexPrefix$_b$. If $b =$ false then $s \neq \texttt{\{} \cdot s'$. 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*$_e$.

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

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

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

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

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

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

Lemma 11 (SourceElement Prefix)**.**
$\forall s, p \in$ RegexPrefix$_b$.

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

Proof. There are two cases:

- $s \in$ *Stmt*$_e$. This holds by Lemma 12.
- $s \in$ *Function*$_e$. Follows directly.

□

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

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

Proof. We have several cases:

- read$(s,\ p,$ *false*$) \in \underline{\texttt{\{}\textit{StmtList'}_e\texttt{\}}}$. Follows since $p \in$ *RegexPrefix*$_b$.
- read$(s,\ p,$ *false*$) \in$ *AssignExpr'*$_e \cdot \texttt{;}$. Follows since $t \cdot \texttt{;} \in$ *RegexPrefix*$_b$ for any t.

- $t = \text{read}(s,\ p,\ \textit{false}) \in \texttt{if} \cdot \underline{\texttt{(}\textit{AssignExpr'}_e\texttt{)}}\ \textit{Stmt}_{e'}$. Since $t = \texttt{if} \cdot \underline{\texttt{(}t'\texttt{)}} \cdot t''$ where $t'' \in \textit{Stmt'}_{e'}$ by induction $t'' \in \textit{RegexPrefix}_b$ and thus $p \cdot t \in \textit{RegexPrefix}_b$.
- $\text{read}(s,\ p,\ \textit{false}) \in \texttt{return}$. Follows since $p \cdot \texttt{return} \in \textit{RegexPrefix}_b$.
- $t = \text{read}(s,\ p,\ \textit{false}) \in \texttt{return} \cdot \textit{AssignExpr'}_e \cdot \texttt{;}$. Since $\texttt{;} \in \textit{RegexPrefix}_b$ then $p \cdot t \in \textit{RegexPrefix}_b$.
- $t = \text{read}(s,\ p,\ \textit{false}) \in \texttt{x} \cdot \texttt{:} \cdot \textit{Stmt'}_e$. Since $t = \texttt{x} \cdot \texttt{:} \cdot t'$ where $t' \in \textit{Stmt'}_e$. Since $p \in \textit{RegexPrefix}_b$ and by induction $t' \in \textit{RegexPrefix}_b$ we have $p \cdot t \in \textit{RegexPrefix}_b$.

□