Sweet.js - Hygienic Macros for JavaScript

Tim Disney
UC Santa Cruz

Nate Faubion

David Herman
Mozilla

1. Introduction

Sweet.js is a new hygienic macro system for JavaScript.

Macros systems have a long history in the design of extensible programming languages going back at least to lisp 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.

Recently the Honu project 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 show how to extend the ideas of Honu for full JavaScript and present additional techniques that target expression based languages.

The design of sweet.js attempts to overcome the following technical challenges:

- a correct implementation of `read` that structures the token stream before expansion begins
- parser class annotation (e.g. `:expr`) in patterns to allow macro authors easier declaration of a macro shape
- operator overloading
- infix macros
- the `invoke` primitive to allow custom parser classes and more powerful matching

2. Overview

TODO: syntax, main features etc...

3. Read

The syntax of JavaScript presents a challenge to correctly implement the critical `read` function. This challenge is not present in Honu because their language is an idealized syntax that misses the problematic interaction of delimiters and regular expression literals.

TODO: motivate `read` with examples etc.

$$\begin{array}{rcl} e & ::= & \texttt{x} \mid \texttt{/x/} \\ & \mid & e\texttt{(}e\texttt{)} \\ & \mid & \texttt{if (}e\texttt{)}e \end{array}$$

3.1 Proof of `read`

We first define a simplified grammar that captures just the essential complexity we want to address, namely the interaction between the division symbol `/` and the regular expression literal `/x/`.

$$\begin{array}{rcl} \textit{Token} & ::= & x \mid / \\ \textit{TokenTree} & ::= & x \mid / \mid /x/ \end{array}$$

We also define $\textit{Expr}'$ with the same productions as *Expr* but defined over $\textit{TokenTree}^*$ instead of $\textit{Token}^*$. So the terminal $/x/$ in the second production of *Expr* is three distinct tokens ($/$, x, and $/$) while it is a single token in $\textit{Expr}'$.

We define the grammar of a $\textit{Program}_e$ as a function from *Token* to *Expr*.

$$\textit{Program}_e :: \textit{Token} \rightarrow \textit{Expr}$$

Theorem 1 (Read is correct)**.**
$\forall s \in \textit{Token}. s \in \textit{Expr}_e$ *iff* $t = \texttt{read}(s,\ p,\ 0), t \in \textit{Expr'}_e$.

Proof.
By showing set inclusion. Details in the appendix. ☐

The grammar we present is a simplified version of the grammar specified in the ES5 specification. The simplification we are making do not affect the read algorithm, and only serve to shorten its presentation. In particular, functions can only have a single parameter, function bodies are not allowed to be empty and a few other minor changes. We only address the = assignment operator and only the / and + binary operators.

Figure 1: Simplified ES Grammar

$\mathit{PrimaryExpression}_{\texttt{x}}$	$::=$	$\texttt{x}$
$\mathit{PrimaryExpression}_{\texttt{/x/}}$	$::=$	$/ \cdot \texttt{x} \cdot /$
$\mathit{PrimaryExpression}_{\texttt{\{x:}e\texttt{\}}}$	$::=$	$\texttt{\{x:}\mathit{AssignmentExpression}_e\texttt{\}}$
$\mathit{PrimaryExpression}_e$	$::=$	$\texttt{(}\mathit{Expression}_e\texttt{)}$
$\mathit{MemberExpression}_e$	$::=$	$\mathit{PrimaryExpression}_e$
$\mathit{MemberExpression}_e$	$::=$	$\mathit{FunctionExpression}_e$
$\mathit{MemberExpression}_{e\texttt{.x}}$	$::=$	$\mathit{MemberExpression}_e\ \texttt{. x}$
$\mathit{CallExpression}_{e\ \texttt{(}e'\texttt{)}}$	$::=$	$\mathit{MemberExpression}_e\ \texttt{(}\mathit{AssignmentExpression}_{e'}\texttt{)}$
$\mathit{CallExpression}_{e\ \texttt{(}e'\texttt{)}}$	$::=$	$\mathit{CallExpression}_e\ \texttt{(}\mathit{AssignmentExpression}_{e'}\texttt{)}$
$\mathit{CallExpression}_{e\texttt{.x}}$	$::=$	$\mathit{CallExpression}_e\ \texttt{. x}$
$\mathit{BinaryExpression}_e$	$::=$	$\mathit{CallExpression}_e$
$\mathit{BinaryExpression}_{e\ \texttt{/}\ e'}$	$::=$	$\mathit{BinaryExpression}_e\ /\ \mathit{BinaryExpression}_{e'}$
$\mathit{BinaryExpression}_{e\ \texttt{+}\ e'}$	$::=$	$\mathit{BinaryExpression}_e + \mathit{BinaryExpression}_{e'}$
$\mathit{AssignmentExpression}_e$	$::=$	$\mathit{BinaryExpression}_e$
$\mathit{AssignmentExpression}_{e\ \texttt{=}\ e'}$	$::=$	$\mathit{CallExpression}_e = \mathit{AssignmentExpression}_{e'}$
$\mathit{ExpressionStatement}_e$	$::=$	$\mathit{AssignmentExpression}_e\ \texttt{;}$ *where lookahead* $\neq$ *{ or* `function`
$\mathit{ReturnStatement}_{\texttt{return ;}}$	$::=$	`return ;`
$\mathit{ReturnStatement}_{\texttt{return}\ e}$	$::=$	`return` [no line terminator here] $\mathit{AssignmentExpression}_e$
$\mathit{Block}_{\texttt{\{}e\texttt{\}}}$	$::=$	$\texttt{\{}\mathit{StatementList}_e\texttt{\}}$
$\mathit{StatementList}_e$	$::=$	$\mathit{Statement}_e$
$\mathit{StatementList}_{e\ e'}$	$::=$	$\mathit{StatementList}_e\ \mathit{Statement}_{e'}$
$\mathit{LabelledStatement}_{\texttt{x:}\ e}$	$::=$	$\texttt{x :}\ \mathit{Statement}_e$
$\mathit{IfStatement}_{\texttt{if (}e\texttt{)}\ e'\ \texttt{else}\ e''}$	$::=$	$\texttt{if (}\mathit{AssignmentExpression}_e\texttt{)}\ \mathit{Statement}_{e'}\ \texttt{else}\ \mathit{Statement}_{e''}$
$\mathit{Statement}_e$	$::=$	Block_e
$\mathit{Statement}_e$	$::=$	$\mathit{ExpressionStatement}_e$
$\mathit{Statement}_e$	$::=$	$\mathit{IfStatement}_e$
$\mathit{Statement}_e$	$::=$	$\mathit{ReturnStatement}_e$
$\mathit{Statement}_e$	$::=$	$\mathit{LabelledStatement}_e$
$\mathit{FunctionDeclaration}_{\texttt{function x (x) \{}e\texttt{\}}}$	$::=$	$\texttt{function x (x) \{}\mathit{SourceElements}_e\texttt{\}}$
$\mathit{FunctionExpression}_{\texttt{function x (x) \{}e\texttt{\}}}$	$::=$	$\texttt{function x (x) \{}\mathit{SourceElements}_e\texttt{\}}$
$\mathit{SourceElement}_e$	$::=$	$\mathit{Statement}_e$
$\mathit{SourceElement}_e$	$::=$	$\mathit{FunctionDeclaration}_e$
$\mathit{SourceElements}_e$	$::=$	$\mathit{SourceElement}_e$
$\mathit{SourceElements}_{e\ e'}$	$::=$	$\mathit{SourceElements}_e\ \mathit{SourceElement}_{e'}$
$\mathit{Program}_e$	$::=$	$\mathit{SourceElements}_e$
$\mathit{Program}$	$::=$	ϵ

Figure 2: Simplified Read Algorithm

$$
\begin{array}{lcl}
\textit{Punctuator} & ::= & \texttt{/} \mid \texttt{+} \mid \texttt{:} \mid \texttt{;} \mid \texttt{=} \mid \texttt{.} \\
\textit{Keyword} & ::= & \texttt{return} \mid \texttt{function} \mid \texttt{if} \mid \texttt{else} \\
\textit{ExpressionPrefix} & ::= & \texttt{=} \mid \texttt{/} \mid \texttt{+} \mid \texttt{return} \\
\textit{DeclarationPrefix} & ::= & \texttt{x} \mid \texttt{/x/} \mid \texttt{else} \mid \texttt{;}
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{read}(/ \cdot x \cdot / \cdot s,\ \epsilon,\ b) & = & /x/ \cdot \texttt{read}(s,\ /x/,\ b) \\
\texttt{read}(/ \cdot x \cdot / \cdot s,\ p \cdot p',\ b) & = & /x/ \cdot \texttt{read}(s,\ p \cdot p' \cdot /x/,\ b) \\
 & & \quad \textit{if } p' \in \textit{Punctuator or Keyword} \\
\texttt{read}(/ \cdot x \cdot / \cdot s,\ p \cdot \texttt{if} \cdot \texttt{(}p'\texttt{)},\ b) & = & /x/ \cdot \texttt{read}(s,\ p \cdot \texttt{if} \cdot \texttt{(}p'\texttt{)} \cdot /x/,\ b) \\
\texttt{read}(/ \cdot x \cdot / \cdot s,\ p \cdot p' \cdot \texttt{function} \cdot \texttt{x} \cdot \texttt{(}q\texttt{)} \cdot \texttt{\{}q'\texttt{\}},\ b) & = & /x/ \cdot \texttt{read}(s,\ p \cdot p' \cdot \texttt{function} \cdot \texttt{x} \cdot \texttt{(}q\texttt{)} \cdot \texttt{\{}q'\texttt{\}} \cdot /x/,\ b) \\
 & & \quad \textit{if } p' \in \textit{DeclarationPrefix} \\
\texttt{read}(/ \cdot x \cdot / \cdot s,\ p \cdot \texttt{\{}q\texttt{\}}^l,\ b) & = & /x/ \cdot \texttt{read}(s,\ p \cdot \texttt{\{}q\texttt{\}}^l \cdot /x/,\ b) \\
 & & \quad \textit{where true} = \texttt{isBlockPrefix}(p,\ b,\ l) \\
 & & \\
\texttt{read}(/ \cdot s,\ p \cdot x,\ b) & = & / \cdot \texttt{read}(s,\ p \cdot x \cdot /,\ b) \\
\texttt{read}(/ \cdot s,\ p \cdot /x/,\ b) & = & / \cdot \texttt{read}(s,\ p \cdot /x/ \cdot /,\ b) \\
\texttt{read}(/ \cdot s,\ p \cdot \texttt{(}p'\texttt{)},\ b) & = & / \cdot \texttt{read}(s,\ p \cdot \texttt{(}p'\texttt{)} \cdot /,\ b) \\
\texttt{read}(/ \cdot s,\ p \cdot p' \cdot \texttt{function} \cdot \texttt{x} \cdot \texttt{(}q\texttt{)} \cdot \texttt{\{}q'\texttt{\}},\ b) & = & / \cdot \texttt{read}(s,\ p \cdot p' \cdot \texttt{function} \cdot \texttt{x} \cdot \texttt{(}q\texttt{)} \cdot \texttt{\{}q'\texttt{\}} \cdot /,\ b) \\
 & & \quad \textit{if } p' \in \textit{ExpressionPrefix} \\
\texttt{read}(/ \cdot s,\ p \cdot \texttt{\{}q'\texttt{\}}^l,\ b) & = & / \cdot \texttt{read}(s,\ p \cdot \texttt{\{}q'\texttt{\}}^l \cdot /,\ b) \\
 & & \quad \textit{where false} = \texttt{isBlockPrefix}(p,\ b,\ l) \\
 & & \\
\texttt{read}(\texttt{(} \cdot s \cdot \texttt{)} \cdot s',\ p,\ b) & = & \texttt{(}t\texttt{)} \cdot \texttt{read}(s',\ p \cdot \texttt{(}t\texttt{)},\ b) \\
\quad \textit{where } s \textit{ contains no unmatched } \texttt{)} & & \quad \textit{where } t = \texttt{read}(s,\ \epsilon,\ \textit{false}) \\
\texttt{read}(\texttt{\{}^l \cdot s \cdot \texttt{\}} \cdot s',\ p,\ b) & = & \texttt{\{}t\texttt{\}}^l \cdot \texttt{read}(s',\ p \cdot \texttt{\{}t\texttt{\}}^l,\ b) \\
\quad \textit{where } s \textit{ contains no unmatched } \texttt{\}} & & \textit{where } t = \texttt{read}(s,\ \epsilon,\ \texttt{isBlockPrefix}(p,\ b,\ l)) \\
\texttt{read}(x \cdot s,\ p,\ b) & = & x \cdot \texttt{read}(s,\ p \cdot x,\ b) \\
\texttt{read}(\epsilon,\ p,\ b) & = & \epsilon
\end{array}
$$

$$
\begin{array}{lcll}
\texttt{isBlockPrefix}(\epsilon,\ \textit{false},\ l) & = & \textit{false} & \\
\texttt{isBlockPrefix}(p \cdot \texttt{/},\ b,\ l) & = & \textit{false} & \\
\texttt{isBlockPrefix}(p \cdot \texttt{+},\ b,\ l) & = & \textit{false} & \\
\texttt{isBlockPrefix}(p \cdot \texttt{:},\ b,\ l) & = & b & \\
\texttt{isBlockPrefix}(p \cdot \texttt{return}^l,\ b,\ l') & = & \textit{true} & \textit{if } l \neq l' \\
\texttt{isBlockPrefix}(p \cdot \texttt{return}^l,\ b,\ l') & = & \textit{false} & \textit{if } l = l' \\
\texttt{isBlockPrefix}(p,\ b,\ l) & = & \textit{true} &
\end{array}
$$

4. Enforestation

The core algorithm introduced by Honu is called *enforestation* which is basically responsible for expanding macros and building a partial syntax tree with enough structure to match on parse classes. Sweet.js implements this algorithm mostly as described with some additions to provide infix macros and invoke pattern classes described below.

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.

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

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

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

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

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

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

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

```
macro (=>) {
    rule infix {
        ($params ...) | { $body ... }
    } => {
        function ($params ...) {
            $body ...
        }
    }
}
var id = (x) => { return x; }
```

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

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

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

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

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

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

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

Mostly straightforward implementation from scheme with some details to handle `var`.

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

- Scheme/Racket
- Honu
- Template Haskell
- Nemerle
- Scala
- Closure

8. Conclusion

A. Read Proof

Theorem 2 (Read Program)**.**
$\forall s. s \in \mathit{Program}_e \Leftrightarrow \mathtt{read}(s,\ \epsilon,\ b) \in \mathit{Program'}_e$

Proof. For the if direction, there are two cases:

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

A similar argument holds for the only if direction. $\square$

Lemma 1 (Read SourceElements)**.**
$\forall s. s \in \mathit{SourceElements}_e \Leftrightarrow \mathtt{read}(s,\ p,\ b) \in \mathit{SourceElements'}_e$

Proof. For the if direction, there are two cases:

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

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

(TODO: need to say something about reading the prefix) By induction $\mathtt{read}(s',\ p,\ b) \in \mathit{SourceElements'}_e$ and by Lemma 2, $\mathtt{read}(s'',\ p \cdot \mathtt{read}(s',\ p,\ b),\ b) \in \mathit{SourceElement'}_{e'}$. Thus $t \in \mathit{SourceElements'}_{e\ e'}$.

The argument is similar for the only if direction. $\square$

Lemma 2 (Read SourceElement)**.**
$\forall s. s \in \mathit{SourceElement}_e \Leftrightarrow \mathtt{read}(s,\ p,\ b) \in \mathit{SourceElement'}_e$

Proof. For the if direction there are two cases:

- $s \in \mathit{SourceElement}_e ::= \mathit{Statement}_e$. This holds by Lemma 4.
- $s \in \mathit{SourceElement}_e ::= \mathit{FunctionDeclaration}_e$. This holds by Lemma 3.

$\square$

Lemma 3 (Read FunctionDeclaration)**.**
$\forall s. s \in \mathit{FunctionDeclaration}_e \Leftrightarrow \mathtt{read}(s,\ p,\ b) \in \mathit{FunctionDeclaration'}_e$

Proof. For the if direction we only have a single case:

$$\begin{aligned} &s \in \mathit{FunctionDeclaration}_{\mathtt{function\ x\ (x)\ \{}e\mathtt{\}}} ::= \\ &\mathtt{function\ x\ (x)\ \{}\mathit{SourceElements}_e\mathtt{\}} \end{aligned}$$

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

$$\begin{aligned} t &= \mathtt{read(function\ x\ (x)\ \{}s'\mathtt{\}},\ p,\ b) \\ &= \mathtt{function\ x\ (x)\ \{}t'\mathtt{\}} \end{aligned}$$

where $t' = \mathtt{read}(s',\ \epsilon,\ \mathit{true})$. **TODO: explain why b = true**. Since by Lemma 1, $t' \in \mathit{SourceElements'}_e$ we have $t \in \mathit{FunctionDeclaration'}_{\mathtt{function\ x\ (x)\ \{}e\mathtt{\}}}$. $\square$

Lemma 4 (Read Statement)**.**
$\forall s. s \in \mathit{Statement}_e \Leftrightarrow \mathtt{read}(s,\ p,\ b) \in \mathit{Statement'}_e$

Proof. For the if direction we have several cases:

- $s \in \mathit{Statement}_e ::= \mathit{Block}_e$. This holds by Lemma 8.
- $s \in \mathit{Statement}_e ::= \mathit{ExpressionStatement}_e$. This holds by Lemma 10.
- $s \in \mathit{Statement}_e ::= \mathit{IfStatement}_e$. This holds by Lemma 5.
- $s \in \mathit{Statement}_e ::= \mathit{ReturnStatement}_e$. This holds by Lemma 7.
- $s \in \mathit{Statement}_e ::= \mathit{LabelledStatement}_e$. This holds by Lemma 6.

$\square$

Lemma 5 (Read IfStatement)**.**
$\forall s. s \in \mathit{IfStatement}_e \Leftrightarrow \mathtt{read}(s,\ p,\ b) \in \mathit{IfStatement'}_e$

Proof. For the if direction we have:

$$\begin{aligned} &s \in \mathit{IfStatement}_{\mathtt{if\ (}e\mathtt{)}\ e'\ \mathtt{else}\ e''} ::= \\ &\mathtt{if\ (}\mathit{AssignmentExpression}_e\mathtt{)}\ \mathit{Statement}_{e'}\ \mathtt{else}\ \mathit{Statement}_{e''} \end{aligned}$$

We have $s = \mathtt{if} \cdot \mathtt{(}s'\mathtt{)} \cdot s'' \cdot \mathtt{else} \cdot s'''$ where $s' \in \mathit{AssignmentExpression}_e$, $s'' \in \mathit{Statement}_{e'}$, and $s''' \in \mathit{Statement}_{e''}$.

$$\begin{aligned} t &= \mathtt{read(if} \cdot \mathtt{(}s'\mathtt{)} \cdot s'' \cdot \mathtt{else} \cdot s''',\ p,\ b) \\ &= \mathtt{if} \cdot \mathtt{(}t'\mathtt{)} \cdot t'' \cdot \mathtt{else} \cdot t''' \end{aligned}$$

where

$$\begin{aligned} t' &= \mathtt{read}(s',\ \epsilon,\ b) \\ t'' &= \mathtt{read}(s'',\ p \cdot \mathtt{if} \cdot \mathtt{(}t'\mathtt{)},\ b) \\ t''' &= \mathtt{read}(s''',\ p \cdot \mathtt{if} \cdot \mathtt{(}t'\mathtt{)} \cdot t'' \cdot \mathtt{else},\ b) \end{aligned}$$

By Lemma 11 $t' \in \mathit{AssignmentExpression'}_e$. By Lemma 4, $t'' \in \mathit{Statement'}_{e'}$ and $t''' \in \mathit{Statement'}_{e''}$. Thus $t \in \mathit{IfStatement'}_{\mathtt{if\ (}e\mathtt{)}\ e'\ \mathtt{else}\ e''}$. $\square$

Lemma 6 (Read LabelledStatement)**.**
$\forall s. s \in \mathit{LabelledStatement}_e \Leftrightarrow \mathtt{read}(s,\ p,\ b) \in \mathit{LabelledStatement'}_e$

Proof. For the if direction we have a single case: $s \in \mathit{LabelledStatement}_{\mathtt{x:}\ e} ::= \mathtt{x}\ \mathtt{:}\ \mathit{Statement}_e$. We have $s = \mathtt{x} \cdot \mathtt{:} \cdot s'$ where $s' \in \mathit{Statement}_e$. Then:

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

By Lemma 4, $\mathtt{read}(s',\ p \cdot \mathtt{x} \cdot \mathtt{:},\ b) \in \mathit{Statement'}_e$ so $t \in \mathit{LabelledStatement'}_{\mathtt{x\ :}\ e}$. **TODO: need to take into account b and the prefix**. $\square$

Lemma 7 (Read ReturnStatement)**.**
$\forall s. s \in \mathit{ReturnStatement}_e \Leftrightarrow \mathtt{read}(s,\ p,\ b) \in \mathit{ReturnStatement'}_e$

Figure 3: Read Algorithm

$$
\begin{array}{lcl}
\texttt{read}(/ \cdot s,\ p \cdot x,\ b) & = & / \cdot \texttt{read}(s,\ p \cdot x \cdot /,\ b) \\
\texttt{read}(/ \cdot s,\ p \cdot /x/,\ b) & = & / \cdot \texttt{read}(s,\ p \cdot /x/ \cdot /,\ b) \\
\texttt{read}(/ \cdot s,\ p \cdot \texttt{function} \cdot \texttt{(}q\texttt{)} \cdot \texttt{\{}q'\texttt{\}},\ b) & = & / \cdot \texttt{read}(s,\ p \cdot \texttt{function} \cdot \texttt{(}q\texttt{)} \cdot \texttt{\{}q'\texttt{\}} \cdot /,\ b) \\
\texttt{read}(/ \cdot s,\ p \cdot p' \cdot \texttt{function} \cdot \texttt{x} \cdot \texttt{(}q\texttt{)} \cdot \texttt{\{}q'\texttt{\}},\ b) & = & / \cdot \texttt{read}(s,\ p \cdot p' \cdot \texttt{function} \cdot \texttt{x} \cdot \texttt{(}q\texttt{)} \cdot \texttt{\{}q'\texttt{\}} \cdot /,\ b) \\
 & & \quad \textit{if } p' \in \textit{ExpressionPrefix} \\
\texttt{read}(/ \cdot s,\ p \cdot \texttt{\{}q'\texttt{\}}^l,\ b) & = & / \cdot \texttt{read}(s,\ p \cdot \texttt{\{}q'\texttt{\}}^l \cdot /,\ b) \\
 & & \quad \textit{where false} = \texttt{isBlockPrefix}(p,\ b,\ l) \\
 & & \\
\texttt{read}(/ \cdot x \cdot / \cdot s,\ \epsilon,\ b) & = & /x/ \cdot \texttt{read}(s,\ /x/,\ b) \\
\texttt{read}(/ \cdot x \cdot / \cdot s,\ p \cdot p',\ b) & = & /x/ \cdot \texttt{read}(s,\ p \cdot p' \cdot /x/,\ b) \quad \textit{if } p' \in \textit{Punctuator or Keyword} \\
\texttt{read}(/ \cdot x \cdot / \cdot s,\ p \cdot \texttt{if} \cdot \texttt{(}p'\texttt{)},\ b) & = & /x/ \cdot \texttt{read}(s,\ p \cdot \texttt{if} \cdot \texttt{(}p'\texttt{)} \cdot /x/,\ b) \\
\texttt{read}(/ \cdot x \cdot / \cdot s,\ p \cdot \texttt{for} \cdot \texttt{(}p'\texttt{)},\ b) & = & /x/ \cdot \texttt{read}(s,\ p \cdot \texttt{for} \cdot \texttt{(}p'\texttt{)} \cdot /x/,\ b) \\
\texttt{read}(/ \cdot x \cdot / \cdot s,\ p \cdot \texttt{while} \cdot \texttt{(}p'\texttt{)},\ b) & = & /x/ \cdot \texttt{read}(s,\ p \cdot \texttt{while} \cdot \texttt{(}p'\texttt{)} \cdot /x/,\ b) \\
\texttt{read}(/ \cdot x \cdot / \cdot s,\ p \cdot \texttt{with} \cdot \texttt{(}p'\texttt{)},\ b) & = & /x/ \cdot \texttt{read}(s,\ p \cdot \texttt{with} \cdot \texttt{(}p'\texttt{)} \cdot /x/,\ b) \\
\texttt{read}(/ \cdot x \cdot / \cdot s,\ p \cdot p' \cdot \texttt{function} \cdot \texttt{x} \cdot \texttt{(}q\texttt{)} \cdot \texttt{\{}q'\texttt{\}},\ b) & = & /x/ \cdot \texttt{read}(s,\ p \cdot p' \cdot \texttt{function} \cdot \texttt{x} \cdot \texttt{(}q\texttt{)} \cdot \texttt{\{}q'\texttt{\}} \cdot /x/,\ b) \\
 & & \quad \textit{if } p' \in \textit{DeclarationPrefix} \\
\texttt{read}(/ \cdot x \cdot / \cdot s,\ p \cdot \texttt{\{}q\texttt{\}}^l,\ b) & = & /x/ \cdot \texttt{read}(s,\ p \cdot \texttt{\{}q\texttt{\}}^l \cdot /x/,\ b) \\
 & & \quad \textit{where true} = \texttt{isBlockPrefix}(p,\ b,\ l) \\
 & & \\
\texttt{read}(\texttt{(} \cdot s \cdot \texttt{)} \cdot s',\ p,\ b) & = & \texttt{(}t\texttt{)} \cdot \texttt{read}(s',\ p \cdot \texttt{(}t\texttt{)},\ b) \\
\quad \textit{where } s \textit{ contains no unmatched } \texttt{)} & & \quad \textit{where } t = \texttt{read}(s,\ \epsilon,\ \textit{false}) \\
\texttt{read}(\texttt{\{}^l \cdot s \cdot \texttt{\}} \cdot s',\ p,\ b) & = & \texttt{\{}t\texttt{\}}^l \cdot \texttt{read}(s',\ p \cdot \texttt{\{}t\texttt{\}}^l,\ b) \\
\quad \textit{where } s \textit{ contains no unmatched } \texttt{\}} & & \textit{where } t = \texttt{read}(s,\ \epsilon,\ \texttt{isBlockPrefix}(p,\ b,\ l)) \\
\texttt{read}(x \cdot s,\ p,\ b) & = & x \cdot \texttt{read}(s,\ p \cdot x,\ b) \\
\texttt{read}(\epsilon,\ p,\ b) & = & \epsilon
\end{array}
$$

$$
\begin{array}{lcll}
\texttt{isBlockPrefix}(\epsilon,\ \textit{false},\ l) & = & \textit{false} & \\
\texttt{isBlockPrefix}(p \cdot \texttt{/},\ b,\ l) & = & \textit{false} & \\
\texttt{isBlockPrefix}(p \cdot \texttt{+},\ b,\ l) & = & \textit{false} & \\
\texttt{isBlockPrefix}(p \cdot \texttt{:},\ b,\ l) & = & b & \\
\texttt{isBlockPrefix}(p \cdot \texttt{return}^l,\ b,\ l') & = & \textit{true} & \textit{if } l \neq l' \\
\texttt{isBlockPrefix}(p \cdot \texttt{return}^l,\ b,\ l') & = & \textit{false} & \textit{if } l = l' \\
\texttt{isBlockPrefix}(p \cdot \texttt{void},\ b,\ l) & = & \textit{false} & \\
\texttt{isBlockPrefix}(p \cdot \texttt{typeof},\ b,\ l) & = & \textit{false} & \\
\texttt{isBlockPrefix}(p \cdot \texttt{in},\ b,\ l) & = & \textit{false} & \\
\texttt{isBlockPrefix}(p \cdot \texttt{case},\ b,\ l) & = & \textit{false} & \\
\texttt{isBlockPrefix}(p \cdot \texttt{delete},\ b,\ l) & = & \textit{false} & \\
\texttt{isBlockPrefix}(p,\ b,\ l) & = & \textit{true} &
\end{array}
$$

Proof. To show the if direction there are two cases:

- $s \in \textit{ReturnStatement}_{\texttt{return ;}} ::= \texttt{return ;}$. Since $s = \texttt{return ;}$ and $\texttt{read}(s,\ p,\ q) = \texttt{return} \cdot \texttt{;} \in \textit{ReturnStatement'}_{\texttt{return ;}}$ we have our result directly.
- $s \in \textit{ReturnStatement}_{\texttt{return}\ e} ::= \texttt{return}\ \textit{AssignmentExpression}_e$. We have $s = \texttt{return} \cdot s'$ where $s' \in \textit{AssignmentExpression}_e$. Then:

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

By Lemma 11, $\texttt{read}(s',\ p \cdot \texttt{return},\ b) \in \textit{AssignmentExpression}_e$ thus $t \in \textit{ReturnStatement'}_{\texttt{return}\ e}$.

□

Lemma 8 (Read Block)**.**
$\forall s. s \in \textit{Block}_e \Leftrightarrow \texttt{read}(s,\ p,\ b) \in \textit{Block'}_e$

Proof. For the if direction we have:

$$s \in \textit{Block}_{\texttt{\{}e\texttt{\}}} ::= \texttt{\{}\textit{StatementList}_e\texttt{\}}$$

so $s = \texttt{\{}s'\texttt{\}}$ where $s' \in \textit{StatementList}_e$.

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

where $t' = \texttt{read}(s',\ \epsilon,\ \textit{true})$. By Lemma 9, $t' \in \textit{StatementList'}_e$ so $t \in \textit{Block'}_{\texttt{\{}e\texttt{\}}}$. □

Lemma 9 (Read StatementList)**.**
$\forall s. s \in \textit{StatementList}_e \Leftrightarrow \texttt{read}(s,\ p,\ b) \in \textit{StatementList'}_e$

Proof. For the if direction we have two cases:

- $s \in \textit{StatementList}_e ::= \textit{Statement}_e$. This follows by Lemma 4.

- $s \in \mathit{StatementList}_{e\ e'} ::= \mathit{StatementList}_e\ \mathit{Statement}_{e'}$. So $s = s' \cdot s''$ where $s' \in \mathit{StatementList}_e$ and $s'' \in \mathit{Statement}_{e'}$. Then,

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

  By induction $\mathtt{read}(s',\ p,\ b) \in \mathit{StatementList'}_e$ and by Lemma 4, $\mathtt{read}(s'',\ p{\cdot}\mathtt{read}(s',\ p,\ b),\ b) \in \mathit{Statement'}_{e'}$. Thus, $t \in \mathit{StatementList'}_{e\ e'}$.

$\square$

Lemma 10 (Read ExpressionStatement)**.** $\forall s.s \in \mathit{ExpressionStatement}_e \Leftrightarrow \mathit{read}(s,\ p,\ b) \in \mathit{ExpressionStatement'}_e$

Proof. For the if direction we have $s \in \mathit{ExpressionStatement}_e ::= \mathit{AssignmentExpression}_e$ `;` so $s = s' \cdot$ `;` where $s' \in \mathit{AssignmentExpression}_e$. Then:

$$\begin{aligned} t &= \mathtt{read}(s' \cdot \mathtt{;},\ p,\ b) \\ &= \mathtt{read}(s',\ p,\ b) \cdot \mathtt{;} \end{aligned}$$

Since by Lemma 11, $\mathtt{read}(s',\ p,\ b) \in \mathit{AssignmentExpression'}_e$ we have $t \in \mathit{ExpressionStatement'}_e$. $\square$

Lemma 11 (Read AssignmentExpression)**.** $\forall s.s \in \mathit{AssignmentExpression}_e \Leftrightarrow \mathit{read}(s,\ p,\ b) \in \mathit{AssignmentExpression'}_e$

Proof. For the if direction we have two cases:

- $s \in \mathit{AssignmentExpression}_e ::= \mathit{BinaryExpression}_e$. This holds by Lemma 12.
- $s \in \mathit{AssignmentExpression}_{e\ \mathtt{=}\ e'}$ by

$$\mathit{CallExpression}_e\ \mathtt{=}\ \mathit{AssignmentExpression}_{e'}$$

  Then $s = s' \cdot \mathtt{=} \cdot s''$ where $s' \in \mathit{CallExpression}_e$ and $s'' \in \mathit{AssignmentExpression}_{e'}$. Then:

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

  By Lemma 13, $\mathtt{read}(s',\ p,\ b) \in \mathit{CallExpression'}_e$ and by induction $\mathtt{read}(s'',\ p \cdot \mathtt{read}(s',\ p,\ b) \cdot \mathtt{=},\ b) \in \mathit{AssignmentExpression'}_{e'}$ so $t \in \mathit{AssignmentExpression'}_{e\ \mathtt{=}\ e'}$.

$\square$

Lemma 12 (Read BinaryExpression)**.** $\forall s.s \in \mathit{BinaryExpression}_e \Leftrightarrow \mathit{read}(s,\ p,\ b) \in \mathit{BinaryExpression'}_e$

Proof. To show the if direction we have three cases:

- $s \in \mathit{BinaryExpression}_e ::= \mathit{CallExpression}_e$. This holds by Lemma 13.
- $s \in \mathit{BinaryExpression}_{e\ \mathtt{/}\ e'}$ by

$$\mathit{BinaryExpression}_e\ \mathtt{/}\ \mathit{BinaryExpression}_{e'}$$

  We have $s = s' \cdot \mathtt{/} \cdot s''$ where $s' \in \mathit{BinaryExpression}_e$ and $s'' \in \mathit{BinaryExpression}_{e'}$. Then:

$$\begin{aligned} t &= \mathtt{read}(s' \cdot \mathtt{/} \cdot s'',\ p,\ b) \\ &= \mathtt{read}(s',\ p,\ b) \cdot \mathtt{/} \cdot \mathtt{read}(s'',\ p \cdot \mathtt{read}(s',\ p,\ b) \cdot \mathtt{/},\ b) \end{aligned}$$

  TODO: since reasons. By induction $\mathtt{read}(s',\ p,\ b) \in \mathit{BinaryExpression'}_e$ and $\mathtt{read}(s'',\ p \cdot \mathtt{read}(s',\ p,\ b) \cdot \mathtt{/},\ b) \in \mathit{BinaryExpression'}_{e'}$ thus $t \in \mathit{BinaryExpression'}_{e\ \mathtt{/}\ e'}$.
- $s \in \mathit{BinaryExpression}_{e\ \mathtt{+}\ e'}$ by

$$\mathit{BinaryExpression}_e\ \mathtt{+}\ \mathit{BinaryExpression}_{e'}$$

  We have $s = s' \cdot \mathtt{+} \cdot s''$ where $s' \in \mathit{BinaryExpression}_e$ and $s'' \in \mathit{BinaryExpression}_{e'}$. Then:

$$\begin{aligned} t &= \mathtt{read}(s' \cdot \mathtt{+} \cdot s'',\ p,\ b) \\ &= \mathtt{read}(s',\ p,\ b) \cdot \mathtt{+} \cdot \mathtt{read}(s'',\ p \cdot \mathtt{read}(s',\ p,\ b) \cdot \mathtt{+},\ b) \end{aligned}$$

  TODO: since reasons. By induction $\mathtt{read}(s',\ p,\ b) \in \mathit{BinaryExpression'}_e$ and $\mathtt{read}(s'',\ p \cdot \mathtt{read}(s',\ p,\ b) \cdot \mathtt{+},\ b) \in \mathit{BinaryExpression'}_{e'}$ thus $t \in \mathit{BinaryExpression'}_{e\ \mathtt{+}\ e'}$.

$\square$

Lemma 13 (Read CallExpression)**.** $\forall s.s \in \mathit{CallExpression}_e \Leftrightarrow \mathit{read}(s,\ p,\ b) \in \mathit{CallExpression'}_e$

Proof. For the if direction we have three cases:

- $s \in \mathit{CallExpression}_{e\ \mathtt{(}e'\mathtt{)}}$ by

$$\mathit{MemberExpression}_e\ \mathtt{(}\mathit{AssignmentExpression'}_{e'}\mathtt{)}$$

  We have $s = s' \cdot \mathtt{(}s''\mathtt{)}$ where $s' \in \mathit{MemberExpression}_e$ and $s'' \in \mathit{AssignmentExpression}_{e'}$. Then

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

  By Lemma 14 we have $\mathtt{read}(s',\ p,\ b) \in \mathit{MemberExpression'}_e$ and my Lemma 11 we have $\mathtt{read}(s'',\ \epsilon,\ \mathit{false}) \in \mathit{AssignmentExpression'}_{e'}$ thus $t \in \mathit{CallExpression'}_{e\ \mathtt{(}e'\mathtt{)}}$.
- $s \in \mathit{CallExpression}_{e\mathtt{.x}} ::= \mathit{CallExpression}_e$ `. x`. Then $s = s' \cdot \mathtt{.} \cdot \mathtt{x}$ where $s' \in \mathit{CallExpression}_e$. Then

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

  By induction $\mathtt{read}(s',\ p,\ b) \in \mathit{CallExpression'}_e$. Thus $t \in \mathit{CallExpression'}_{e\ \mathtt{.x}}$.

$\square$

Lemma 14 (Read MemberExpression)**.** $\forall s.s \in \mathit{MemberExpression}_e \Leftrightarrow \mathit{read}(s,\ p,\ b) \in \mathit{MemberExpression'}_e$

Proof. For the if direction we have three cases:

- $s \in \mathit{MemberExpression}_e$ ::= $\mathit{PrimaryExpression}_e$. This follows from Lemma **??**.
- $s \in \mathit{MemberExpression}_e$ ::= $\mathit{FunctionExpression}_e$. This follows from Lemma **??**.
- $s \in \mathit{MemberExpression}_{e\texttt{.x}}$ ::= $\mathit{MemberExpression}_e$ `. x`. We have $s = s' \cdot \texttt{.x}$ where $s \in \mathit{MemberExpression}_e$. Then

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

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

$\square$

Lemma 15 (Reading a Unit)**.**
$s \in \mathit{ReadUnit} \Rightarrow$

$$\mathit{read}(s{\cdot}s',\ p,\ b) = \mathit{read}(s,\ p,\ b){\cdot}\mathit{read}(s',\ p{\cdot}\mathit{read}(s,\ p,\ b),\ b)$$

Lemma 16 (Reading the end of an expr)**.**
$s \in \mathit{Expr}_e \Rightarrow p \cdot \mathit{read}(s,\ p,\ b) \in \mathit{DivPrefix}$

Lemma 17. $s \in \mathit{Expr}_e \Rightarrow s \in \mathit{ReadUnit}$

Lemma 18 (Read Expr)**.**
$\forall s,\ p.\ p \notin \mathit{DivPrefix}$
$s \in \mathit{Expr}_e \Leftrightarrow \mathit{read}(s,\ p,\ b) \in \mathit{Expr'}_e$

Proof. By cases of Expr_e:

- Case $\mathit{Expr}_{e\ \texttt{/}\ e'}$ ::= Expr_e `/` $\mathit{Expr}_{e'}$. From this $s = s' \cdot \texttt{/} \cdot s''$ where $s' \in \mathit{Expr}_e$ and $s'' \in \mathit{Expr}_{e'}$. So

$$\begin{array}{rcl} t & = & \texttt{read}(s' \cdot \texttt{/} \cdot s'',\ p,\ b) \\ & = & \texttt{read}(s',\ p,\ b) \cdot \texttt{read}(\texttt{/} \cdot s'',\ p \cdot \texttt{read}(s',\ p,\ b),\ b) \\ & & \text{(by Lemma 15 since } s' \in \mathit{ReadUnit}\text{)} \\ & = & \texttt{read}(s',\ p,\ b) \cdot \texttt{/} \cdot \texttt{read}(s'',\ p \cdot \texttt{read}(s',\ p,\ b) \cdot \texttt{/},\ b) \\ & & \text{(since by Lemma 16, } \texttt{read}(s',\ p,\ b) \in \mathit{DivPrefix}\text{)} \end{array}$$

  By induction $\texttt{read}(s',\ p,\ b) \in \mathit{Expr'}_e$ and $\texttt{read}(s'',\ p \cdot \texttt{read}(s',\ p,\ b) \cdot \texttt{/},\ b) \in \mathit{Expr'}_{e'}$ so $t \in \mathit{Expr'}_{e\ \texttt{/}\ e'}$.

$\square$