UNIVERSITY OF CALIFORNIA
SANTA CRUZ

# HYGIENIC MACROS FOR JAVASCRIPT

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

**Tim Disney**

September 2015

The Dissertation of Tim Disney
is approved:

# Table of Contents

# List of Figures

**Abstract**

Hygienic Macros for JavaScript

by

Tim Disney

Languages like Lisp, Scheme, and Racket have shown that powerful and expressive macro systems can give programmers the ability to grow their own language. Unfortunately, in languages with syntax like JavaScript, macros have had less success, due in part to the difficulty of integrating macro expansion and complex grammars.

This dissertation describes sweet.js, a hygienic macro system for JavaScript that fixes long standing challenges in lexing JavaScript and provides expressive pattern matching that allows macros to manipulate the complex grammar of JavaScript. With sweet.js programmers can experiment with syntax extensions for future versions of JavaScript and craft domain specific languages.

## Acknowledgments

Many thanks to my advisor, Cormac Flanagan, whose wisdom and guidance has proved so invaluable over the years.

Thanks to my dissertation committee, Martin Abadi and Dave Herman, for providing their insight and encouragement. In particular, thanks to Dave for allowing me to work on sweet.js when I was a Mozilla intern (and for providing the name and logo!).

My fellow labmates at UC Santa Cruz, Tom Austin, Jeaheon Yi, Caitlin Sadowski, Chris Lewis, Tommy Schmitz, Chris Schuster, and Dustin Rhodes where a constant source of inspiration and wonderful conversations.

Seeing the open source community surrounding sweet.js has been humbling. Nathan Faubion, James Long, Paul Taylor, and Paul Stansifer, to name just a few, contributed ideas, code, macros, documentation, and bug reports to the project. Sweet.js could not have been built without them.

# Chapter 1

# Introduction

A key goal in the development of programming languages is abstraction. Abstraction facilities allow programmers to more easily reason about code; procedural abstraction (e.g., functions) helps programmers reason at a high level about program behavior while data abstraction (e.g., objects or ADTs) provides similar reasoning benefits on the structure of data.

Syntactic abstraction facilities aim to provide similar reasoning benefits to the syntax of programming languages. While virtually all modern languages have some degree of procedural and data abstraction, syntactic abstraction is more rare.

Macro systems are one kind of syntactic abstraction facility that allow programmers to define new syntax in their language. Macros are particularly important for extensible languages as a means of enabling a language that can grow [Steele, 1999]. In a language with macros, programmers can build syntax features that are specific to their domain (i.e., macros can be used to implement domain specific languages) along with creating broadly usable syntactic forms that could be standardized in future versions of the language.

Expressive macro systems have a long history in the design of extensible programming languages going back to Lisp and Scheme [Foderaro et al., 1983, Kohlbecker and Wand, 1987]. The Scheme community in particular introduced *hygienic* [Clinger, 1991, Dybvig et al., 1992] macros that prevent binding clashes from occurring during macro expansion.

While expressive macro systems have been very successful in languages with s-expressions (for example, Racket [Matthew Flatt and PLT, 2010] provides extensive support for constructing fully fledged languages [Tobin-Hochstadt et al., 2011] via its macro system), in languages with more complex syntax, macros have had less success. In part, this lack of success has been due to the difficulty in building expressive hygienic macro systems for such languages.

JavaScript in particular presents a unique challenge for macros. JavaScript is a challenge for macro systems due to its complex grammar and ambiguities in the lexing and parsing stages.

In this dissertation I present a hygienic macro system for JavaScript, sweet.js, that addresses many of the challenges in building a macro system for languages with a complex grammar. There are three primary design goals that sweet.js address: resolving the ambiguities in lexing and parsing JavaScript, enabling expressive macro definitions for the complex grammar of JavaScript, and building a path towards full syntactic abstraction.

## 1.1 Resolving Lexing and Parsing Ambiguities

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

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

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

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

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

As a first step to designing a Scheme-like macro system for a language like JavaScript, it is necessary to introduce a read step into the compiler pipeline. However, the lexical structure of mainstream languages can be surprisingly subtle. In JavaScript, implementing a correct reader is complicated 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 forced to be intertwined; a JavaScript lexer cannot run in isolation. Rather, the parser calls out to the lexer from a given grammatical context with a flag to indicate if the lexer should accept a regular expression or a divide operator, and the input character stream is lexed accordingly. So if the parser is in a context that accepts a regular expression, the characters "`/x/`" will be lexed into the single lexeme $\underline{\texttt{/x/}}$ otherwise it will lex into the individual lexemes $\underline{\texttt{/}}$, $\underline{\texttt{x}}$, and $\underline{\texttt{/}}$.

$$\begin{tikzcd} \textit{lexer} \arrow[r, shift left, "\textit{Lexeme}^*"'] & \textit{parser} \arrow[l, shift left, "\textit{feedback}"'] \arrow[r, "\textit{AST}"] & {} \end{tikzcd}$$

An important first step to implement a macro system for JavaScript is to remove the dependency between the lexing and parsing stages. Sweet.js includes a separate reader that converts a sequence of lexemes into a sequence of tokens (tokens in this setting are a little analogous to s-expressions in that they can be nested) without feedback from the parser.

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

The reader must account for the ambiguities caused by the symbol `/`. In this dissertation I present a novel solution to resolving this lexing ambiguity in JavaScript that enables the clean separation of the JavaScript lexer and parser. The key idea is to record sufficient history information in the form of tokens in order to correctly decide whether to parse `/x/g` as a regular expression or as division operators (as in `4.0/x/g`). Surprisingly, this history information needs to be remembered from arbitrarily far back in the token stream. Thankfully, the reader can be implemented efficiently by collapsing this history into a single boolean plus at most five tokens of lookbehind. An algorithm for this solution along with a proof that it does in fact correctly resolve ambiguities in the language is presented in Chapter 3.

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

## 1.2 Declarative Macro Definitions

The second challenge for building an expressive macro system is declarative macro definitions. To easily build and reason about a macro, macro authors need declarative syntax. In Lisp or Scheme, since the base language is just s-expressions, it is straightforward to provide declarative pattern matching for s-expressions, a technique known as macro-by-example [Kohlbecker and Wand, 1987]. In a language like JavaScript however, the base language is much more complex and the straightforward techniques of macro-by-example are not sufficient. To address this challenge I extend the *enforestation* technique of Rafkind [Rafkind, 2013, Rafkind and Flatt, 2012], which provides robust and declarative pattern matching for syntactic forms in the base language.

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

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

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

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

This approach, called *enforestation*, was pioneered by Honu [Rafkind and Flatt, 2012], which I adapt here for JavaScript[1].

## 1.3 Full Syntactic Abstraction

A key design goal that sweet.js strives for and partially achieves is *full syntactic abstraction*. This goal stems from the desire to enable programmers to extend any syntactic form in JavaScript in a transparent manner. In particular, a programmer who uses sweet.js should not be able to textually distinguish between a builtin syntactic form and a syntactic form that is implemented via a macro. This design goal distinguishes sweet.js from other macro systems such as Template Haskell [Sheard and Jones, 2002] or Rust [Matsakis and Klock, 2014] that use special syntax at macro invocation sites.

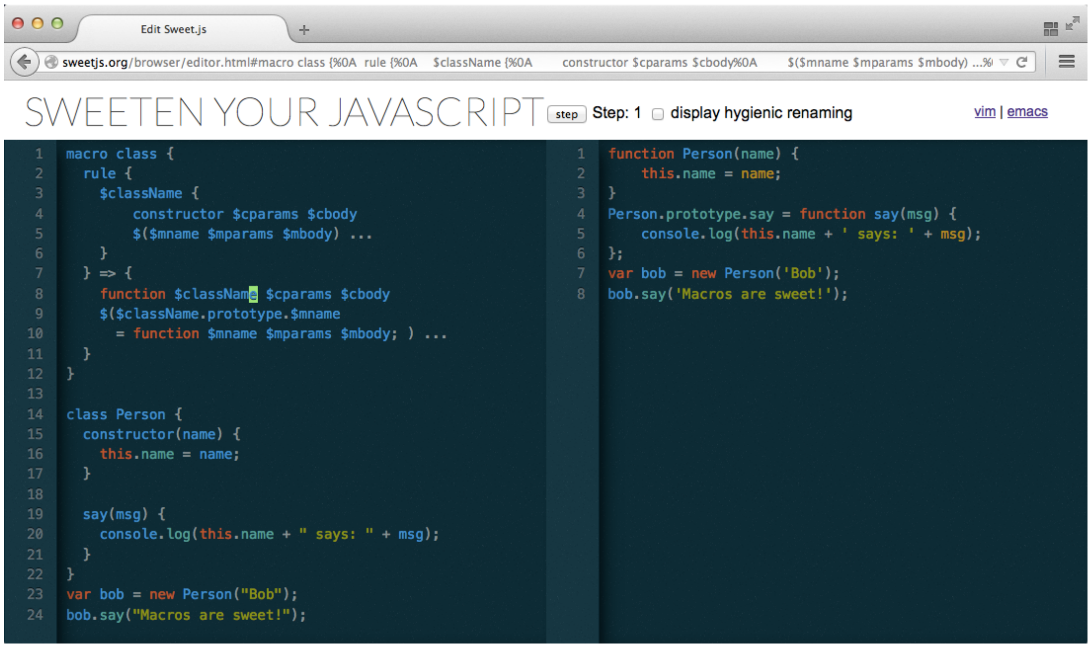
In addition, full syntactic abstraction means that any syntactic form in the base language should be implementable via a macro. This goal has motivated the implementation of *infix macros* (which allow macros to be written that match syntax before the macro identifier). Infix macros enable, for example, the implementation of the arrow function that is standardized in the latest version of JavaScript, ES2015 [International, 2015].

## 1.4 Implementation

Sweet.js is implemented in JavaScript, is publicly available, and is open source. Sweet.js is a compiler that takes source code written with sweet.js macros and generates source code with the macros expanded away that can be run in any JavaScript environment. The project web page[2] includes an interactive browser-based editor that makes it simple to try out writing macros without requiring any installation. Figure 1.1 shows the editor in action; a macro implementing classes is being edited in the left pane and the right pane continually shows the expanded output. There is already an active community using sweet.js, for example, to implement significant features from the upcoming ES2015 version of JavaScript [Long] or implement pattern matching in JavaScript [Faubion].

[1]The syntax of Honu is similar to JavaScript but does not support regular expression literals, which simplifies the Honu reader.
[2]http://sweetjs.org

**Figure 1.1: The Sweet.js Editor**


Edit Sweet.js
sweetjs.org/browser/editor.html#macro class {%0A rule {%0A $className {%0A constructor $cparams $cbody%0A $($mname $mparams $mbody) ...%
SWEETEN YOUR JAVASCRIPT
step
Step: 1
display hygienic renaming
vim | emacs

```
macro class {
  rule {
    $className {
        constructor $cparams $cbody
        $($mname $mparams $mbody) ...
    }
  } => {
    function $className $cparams $cbody
    $($className.prototype.$mname
      = function $mname $mparams $mbody; ) ...
  }
}

class Person {
  constructor(name) {
    this.name = name;
  }

  say(msg) {
    console.log(this.name + " says: " + msg);
  }
}
var bob = new Person("Bob");
bob.say("Macros are sweet!");
```

```
function Person(name) {
    this.name = name;
}
Person.prototype.say = function say(msg) {
    console.log(this.name + ' says: ' + msg);
};
var bob = new Person('Bob');
bob.say('Macros are sweet!');
```

Sweet.js has been developed in the open with over 30 contributors providing documentation, bug reports, bug fixes, and significant features. At the time of this writing it has been starred on GitHub over 3,000 times, forked over 100 times, and the Node.js package for sweet.js[3] has been downloaded over 22,000 times.

## 1.5 Contributions

The key contributions of this dissertation are:

- a novel `read` algorithm that resolves long standing ambiguities in the lexical structure of JavaScript
- a proof of correctness for the `read` algorithm
- a formalization of hygienic macro expansion with enforestation for a simplified JavaScript-like language
- an implementation of a hygienic macro system for full JavaScript called sweet.js
- an evaluation of sweet.js via two language extensions:
- a contract system for JavaScript
- an extension of the ES2015 Proxy API that supports primitive operations and primitive values

## 1.6 Outline

In Chapter 2 I present a user-centered overview of sweet.js and demonstrate how to build a variety of macros using the system. Chapter 3 shows how to separate the JavaScript lexer and parser. Chapter 4 presents a non-hygienic formal semantics for macro expansion while Chapter 5 describes hygiene and extension the formal semantics to maintain hygiene. Chapters 6 and 7 describe using macros to build a contract system and extend JavaScript to support virtual values respectively. Chapter 8 summarizes the design goals and lessons learned in sweet.js, reviews related work, and concludes.

[3]https://www.npmjs.com/package/sweet.js

# Chapter 2

# The Sweet.js Macro System

In this chapter I present the programmer's view of using sweet.js to build a variety of syntax abstractions. Later chapters will discuss the implementation details of sweet.js. As motivation, I use a series of macros that implement features that have either been standardized in ES2015 or are being proposed for future standardization. Some of the macros covered in this chapter are also available as a robust implementation in the es6-macros project [Long] created by James Long.

Sweet.js provides two kinds of macros: *rule* and *case* macros (respectively analogous to the `syntax-rules` [Clinger, 1991] and `syntax-case` [Dybvig et al., 1992] Scheme forms). Rule macros are more straightforward to explain and I will discuss them first.

Rule macros are based on the macro-by-example tradition of declarative macro definitions. A rule macro declaration allows a programmer to specify an example syntax *pattern* to match against and an example syntax *template* into which the matched syntax is transformed.

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

To a first approximation (this will grow as I work through various examples) the pattern language contains:

- `$variable` – any identifier beginning with a `$` is a pattern variable and matches a single token, binding it to the variable name

- `<pattern> ...` – match zero or more of `<pattern>`
- `<pattern> (,) ...` – match zero or more of `<pattern>` where a `,` separates each repetition
- everything else matches literally

For the purposes of a pattern variable, a token includes both the "standard" tokens (e.g., an identifier such as `foo` or a string literal like `"macros are sweet!"`) but also delimiter matched groups of tokens. The source `(24 + 42)` counts as a single token for binding to a single pattern variable. Treating delimiter groups as tokens is very useful for expressive macros and will be discussed further in Chapter 3.

## 2.1 Implementing Class

JavaScript is a prototype-based object oriented system rather than the more traditional class-based system of most other OO languages. While the prototype-based approach is powerful and can simulate traditional class patterns, the syntax is unintuitive. For example, the syntax used to define a `Cat` object with two methods is as follows:

```
function Cat(name, age) {
    this.name = name;
    this.age = age;
}
Cat.prototype.canHaz = function(haz) {
    return this.name " can haz " + haz;
}
Cat.prototype.toString = function() {
    return "Cat " + this.name + ", age: " + this.age;
}
```

The ES2015 standard introduces a more intuitive `class` form for object declarations patterns like this example. Using `class` the `Cat` definition becomes:

```
class Cat {
    constructor (name, age) {
        this.name = name;
        this.age = age;
    }
    canHaz (haz) {
        return this.name " can haz " + haz;
    }
    toString () {
        return "Cat " + this.name + ", age: " + this.age;
    }
}
```

As a first step to implement the `class` form with a macro, here is a basic implementation that only supports the constructor function:

```
macro class {
    rule {
        $name {
            constructor ($params (,) ...) { $body ... }
        }
    } => {
        function $name ($params (,) ...) { $body ... }
    }
}
```

Using this macro definition, the expansion of a `Cat` class is shown in Figure 2.1. In this example the pattern variable `$name` matches `Cat`, the pattern `constructor` matches the literal token `constructor`, the parameters `name` and `age` are bound to the repeated pattern variable `$params`, and all of the tokens inside of the constructor body are matched with `$body`.

Extending the `class` macro to handle methods requires the ability to match on a repeated group of patterns, which is handled by the grouping `$()` pattern.

```
macro class {
    rule {
        $name {
            constructor ($params (,) ...) { $body ... }
            $(
            $mname ($mparams (,) ...) { $mbody ... }
            ) ...
        }
```

**Figure 2.1: Class Expansion**

```
class Cat {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }
}
```

expands to ⇒

```
function Cat(name, age) {
    this.name = name;
    this.age = age;
}
```

```
    } => {
        function $name ($params (,) ...) { $body ... }
        $(
        $name.prototype.$mname = function $mname($mparams (,) ...) {
            $mbody ...
        };
        ) ...
    }
}
```

The use of the repeated grouping pattern tells sweet.js to match zero or more method declarations and then expand to code where each method is assigned to the constructor function's prototype.

ES2015 classes also support an optional inheritance pattern by extending the prototype of a base object. The syntax for this feature is `class Cat extends Animal`. The `class` macro now uses two rules, one to handle the `extends` case and one to handle the normal case.

```
macro class {
    rule {
        $name extends $base {
            constructor ($params (,) ...) { $body ... }
            $(
            $mname ($mparams (,) ...) { $mbody ... }
            ) ...
        }
    } => {
        function $name ($params (,) ...) {
            $body ...
        }
        $name.prototype = Object.create($base.prototype);
        $(
        $name.prototype.$mname = function $mname($mparams (,) ...) {
            $mbody ...
        };
        ) ...
    }

    rule {
        $name {
            constructor ($params (,) ...) { $body ... }
            $(
            $mname ($mparams (,) ...) { $mbody ... }
            ) ...
        }
    } => {
     // as before...
    }
}
```

When a macro is invoked, the expander attempts to match each rule in the macro's definition in a top-down order. In this example, the `class` macro will first try to match syntax that includes the `extends` literal and if that fails to match, it will attempt the second rule. If the macro matched the `extends` rule, it would expand to code that wires the prototype chain by using the built in function `Object.create()`, which creates a new object with its prototype set to the first argument (the optional second argument of `create` sets initial property values for the new object and is not needed here).

The final feature of `class` we would like to implement is the `super` keyword, which provides a reference to the extended object. Properly implementing this feature requires some advanced features of sweet.js, so will be discussed later in this chapter.

## 2.2 Rest Rules

ES2015 also introduces quality-of-life syntax improvements to function declarations. A function in JavaScript can be invoked with any number of arguments irrespective of the number of parameters declared in the function definition. In ES5 and before the "extra" arguments are available in the `arguments` implicit parameter, which is an "array-like" object[1] that contains each argument. ES2015 addresses this pattern by adding the rest ... operator on function declarations that allows programmers to name both individual parameters and then the remaining array of the "rest" of the arguments:

```
function foo(x, y, ...args) {
    args.shift();
}
```

To implement rest via a macro, we need to use two new features: let bound macros and escape literals:

```
let function = macro {
    rule {
        $name ($[...] $restName) {
            $body ...
        }
    } => {
        function $name () {
            var $restName = Array.prototype.slice.call(arguments);
            $body ...
        }
    }
```

[1]array-like in the sense that the object provides access to its elements via indexed array notation but is missing many of the normal array methods.

```
}
```

The normal macro declaration form (`macro name { /* .. */ }`) is recursive; it binds the macro's name in its template. This binding behavior is useful for implementing recursive macro definitions but causes a problem when we want to extend the definition of existing syntax forms like `function`, since using a normal macro declaration form for the above `function` macro would result in an infinite expansion loop. Instead, we can use the let binding form which does not bind the identifier `function` to the macro definition in the template.

In addition, the rest ellipses ... conflicts with the sweet.js repetition ellipses so we need a way to match the syntax "..." literally. The pattern `$[...]` allows us to literally match the rest ellipses.

## 2.3 Rest Case

Astute readers will note that the macro we have so far only handles the case where the rest argument is the only parameter, but obviously the feature should allow the definition of multiple named parameters along with the rest parameter.

To handle this case we need some additional power that rule macros do not easily provide us. In particular, we need to know the number of non-rest parameters in the declaration so that the rest parameter can contain only the left over arguments. To be concrete, we want the expansion shown in Figure 2.2.

To get the number of parameters we can use *case* macros. Unlike rule macros, the body of a case macro is JavaScript code that is evaluated at macro invocation time and can manipulate and create new syntax. A case macro allows us to programmatically inspect the number of parameter tokens and create the appropriate syntax.

Case macros look similar to rule macros with a few important differences. First, the patterns also match the macro name instead of just the syntax that comes after it. In our example, it is not necessary to bind the macro name so it can be ignored with the wildcard `_` pattern (which matches any token but does not bind it to a name).

Templates are now created with the `#{...}` form. The `#{...}` form creates an array of *syntax objects* using any pattern bindings that are in scope (i.e., were matched by the pattern).

**Figure 2.2: Rest Expansion**

```
function foo(x, y, ...args) {
    // ...
}
```

expands to ⇒

```
function foo(x, y) {
    // drop the first two elements from arguments
    var args = Array.prototype.slice.call(arguments, 2);
    // ...
}
```

A *syntax object* is the data representation that sweet.js uses to keep track of tokens along with their lexical context (a lexical context tracks binding information and is described in Chapter 5). For example, the syntax object representation of the identifier `foo` looks something like:

```
{
    token: {
        // '3' is the token type of identifiers
        type: 3,
        value: "foo",
        lineNumber: 1
    },
    context: {
        // binding information...
    }
}
```

The following functions create a syntax object for the appropriate kind of token:

- `makeValue(val, stx)` – `val` can be a `boolean`, `number`, `string`, or `null`/`undefined`
- `makeRegex(pattern, flags, stx)` – `pattern` is the string representation of the

regex pattern and `flags` is the string representation of the regex flags

- `makeIdent(val, stx)` – `val` is a string representing an identifier
- `makePunc(val, stx)` – `val` is a string representing a punctuation (e.g. `=`, `,`, `>`, etc.)
- `makeDelim(val, inner, stx)` – `val` represents which delimiter to make and can be either `"()"`, `"[]"`, or `"{}"` and `inner` is an array of syntax objects for all of the tokens inside the delimiter.

The last `stx` argument to each of these function is a syntax object from which the freshly created syntax object takes its lexical context. This syntax argument is used to "bend" hygiene and I will discuss it a bit later. For now, the common pattern is to use `#{here}` when using one of these syntax creation functions (e.g., in `makeValue(42, #{here})`).

When using syntax object creation functions, it is convenient to refer to them in `#{}` templates. To do this sweet.js provides the construct `letstx`, which binds syntax objects to pattern variables:

With all that preamble out of the way, the case macro we need is the following:

```
let function = macro {
    case {_
        $name ($params (,) ...  $[...] $restName) {
            $body ...
        }
    } => {
        // find the number of parameters
        var paramCount = #{$params ...}.length;

        // letstx creates bindings to syntax objects that
        // can be useed in the following templates. Here we
        // create a numeric literal syntax object for the
        // the number of parameters and bind it to $index
        letstx $index = [makeValue(paramCount, #{here})];

        // return the syntax array generated by the following
        // template. Note that $index is bound in the template
        // the the numeric literal syntax object created above
        return #{
            function $name ($params (,) ...) {
                var $restName = Array.prototype.slice.call(arguments, $index);
                $body ...
            }
        }
    }
}
```

In this macro `#{$params ...}` is an array where each element is a parameter syntax object. By checking the `length` of this array we get the total number of parameters and then then use the number of parameters to create a new numeric literal syntax object. Then in the template the length is used as the starting index when slicing `arguments`.

## 2.4 Default Arguments

Default arguments are another addition in ES2015 that allow programmers to specify the default value of parameters when they are not supplied during function invocation.

```
function foo(x = 40 + 2, y = 100) {
    // ...
}
```

In order to properly implement default arguments as a macro, it is necessary to match on the default initialization for each parameter. However, the initialization is an undelimited expression (such as `40 + 2`), which is difficult to properly match against with the pattern language shown so far.

Sweet.js addresses this difficulty with *pattern classes* that specify the JavaScript grammar production that a pattern variable should match against. Pattern classes are attached to pattern variables with the syntax `:<class>`. For example, the pattern `$var:expr` will match an expression and bind the resulting match to the pattern variable `$var`.

With pattern classes we are now in a position to implement default arguments:

```
let function = macro {
    rule {
        $name ( $($param = $init:expr) (,) ... ) {
            $body ...
        }
    } => {
        function $name ( $param (,) ... ) {
            $($param = $param !== undefined ? $param : $init;) ...
            $body ...
        }
    }
}
```

The key idea of this macro is that when the function is invoked, each undefined parameter is set to its default value.

## 2.5 Putting Together Functions

Up until now, while our ES2015 function extensions have illustrated the core technique, they are not very robust macros. The default parameters extension does not support normal parameters (every parameter must be in the form `$param = $init:expr`) and attempting to combine the default parameters extension with the rest parameter extension would lead to lots of repeated code. To address these difficulties we need a way of separating out the essential concerns of each extension to combine them in a clean way.

The first issue we need to handle is extending default parameters to deal with both default and normal parameters in a separable way. The sweet.js feature that allows us to do this separation is the `invoke` pattern class, which allows a macro to invoke another macro during pattern matching. Essentially `invoke` inserts another macro into the token stream allowing some of the pattern matching logic to be moved to the invoked macro.

We can use `invoke` to move the parameter pattern matching logic into a `default_param` macro:

```
macro default_param {
    rule { $param = $init:expr } => { $param }
    rule { $param } => { $param }
}

let function = macro {
    rule {
        $name ( $p:invoke(default_param) (,) ... ) {
            $body ...
        }
    } => {
        function $name ($p (,) ...) {
            $body ...
        }
    }
}
```

If a pattern class matches an in-scope macro, then an implicit invoke is performed. This means that the `function` macro can be simplified to:

```
let function = macro {
    rule {
        $name ( $p:default_param (,) ... ) {
            $body ...
        }
    } => {
        function $name ($p (,) ...) {
            $body ...
        }
    }
}
```

While this macro now separates out the pattern matching logic, we still need a way to refer to both the parameter name and the initializer in the macro template. Referencing the name and initializer is done via `macroclass`, a feature built on top of `invoke`. Using `macroclass` a programmer can define patterns where each sub-pattern can be referenced via concatenation. For example, the default parameter pattern can be defined as:

```
macroclass default_param {
    pattern {
        rule { $param = $init:expr }
    }
}
```

Then a pattern `$p:default_param` can refer to the `$param` sub-pattern via `$p$param` and `$init` via `$p$init`. A `macroclass` can define multiple patterns (matched top down just like normal macros) and a pattern can set missing sub-patterns via `with`. So, a `default_param` that matches both a default parameter and a normal parameter can be defined as:

```
macroclass default_param {
    pattern {
        rule { $param = $init:expr }
    }
    pattern {
        rule { $param }
        with $init = #{undefined}
    }
}
```

Here the normal parameter pattern sets the `$init` sub-pattern to the `undefined` token. Now our `function` macro can use the sub-patterns of `default_param` to expand to the appropriate parameter setting behavior (an expansion of a function with mixed parameters is shown in Figure 2.3.):

```
let function = macro {
    rule {
        $name ( $p:default_param (,) ... ) {
            $body ...
        }
    } => {
        function $name ($p$param (,) ...) {
            $($p$param = $p$param !== undefined
              ? $p$param
              : $p$init;) ...
            $body ...
        }
```

**Figure 2.3: Standard and Default Parameters Expansion**

```
function foo(x, y = 100 + 200) {
    // ...
}
```

expands to ⇒

```
function foo(x, y) {
    x = x !== undefined ? x : undefined;
    y = y !== undefined ? y : 100 + 200;
    // ...
}
```

```
    }
}
```

Now that we have a robust implementation of default parameters that separates the parameter matching we can more easily add rest arguments. As before, we need to switch to using a case macro in order to count the number of parameters. In addition, the rest argument is optional so we need a way of handling both function with and without a rest argument. We can separate out the optional matching into an `opt_rest` macro:

```
macro opt_rest {
    rule { $[...] $rest } => { $rest }
    rule {} => {}
}
```

The `opt_rest` macro works by first attempting to match the pattern `$[...] $rest`. If this first case fails, the empty pattern is attempted. Since the empty pattern will always match, the `opt_rest` macro expands into an empty template.

Now it is straightforward to implement our function macro that supports both default parameters and rest arguments.

```
let function = macro {
    case {_
```

```
        $name ( $p:default_param (,) ... $rest:opt_rest ) {
            $body ...
        }
    } => {
        var restStx = #{};
        var paramCount = #{$p$param ...}.length;
        letstx $index = [makeValue(paramCount, #{here})];
        if (#{$rest}.length > 0) {
            restStx = #{var $rest = Array.prototype.slice.call(arguments, $index);}
        }
        letstx $restStx = restStx;
        return #{
            function $name ($p$param (,) ...) {
                $restStx
                $($p$param = $p$param !== undefined
                  ? $p$param
                  : $p$init;) ...
                $body ...
            }
        }
    }
}
```

The key idea is to set `$restStx` to the appropriate `arguments` manipulating code only if `$rest` matched against a rest argument.

## 2.6 Arrow Functions

ES2015 arrow functions present sweet.js with an additional challenge to the syntax extension we have covered so far. Arrow functions are a concise function expression definition form. For example, a simple `add` function can be defined as:

```
var add = (x, y) => x + y
// syntactic sugar for
// var add = function(x, y) { return x + y }
```

The macros we have described so far are prefix macros: the macro identifier appears before the syntax that it matches. This pattern mirrors most of the standard JavaScript forms (e.g., `function` and `if` begin with an identifying keyword) and allows a macro system to process macros left-to-right.

While prefix macros are sufficient for many syntax forms, other forms such as ES2015 arrow functions require additional flexibility.

We can do this matching of syntax before a macro name with *infix macros*. Defining an infix macro is similar to standard macro except we add the `infix` keyword after each `rule` or `case` keyword and use the pipe operator `|` in the pattern to represent where the macro name should appear (the pipe can be thought of as a cursor into the token stream). When the macro name is encountered during macro expansion, patterns to the left of the `|` are matched against syntax to the left of the macro name and to the right respectively.

With infix macros we can handle all the possible cases of ES2015 arrow functions with only four different rules:

```
macro => {
    // (x, y) => { return x + y; }
    rule infix { ( $params (,) ... ) | { $body ... } } => {
        function ($params (,) ...) {
            $body ...
        }.bind(this)
    }
    // x => { return x; }
    rule infix {  $param | { $body ... } } => {
        function ($param) {
            $body ...
        }.bind(this)
    }
    // (x, y) => x + y
    rule infix { ( $params (,) ... ) | $body:expr } => {
        function ($params (,) ...) {
            return $body;
        }.bind(this)
    }
    // x => x + y
    rule infix { $param | $body:expr } => {
        function ($param ) {
            return $body;
        }.bind(this)
    }
}
```

The use of `.bind(this)` is due to the fact that arrow functions in ES2015 do not define a local binding to `this`. Instead `this` should resolve to the binding in the arrow function's surrounding scope. The ES5 `bind` method on functions allows us to adjust the binding of `this`, which simulates a missing local definition of `this` inside arrow functions.

Infix macros illustrate a guiding principle in the design of sweet.js—namely the ability to build any of the syntax forms of JavaScript as a macro (including forms introduced in ES2015 such as arrow functions).

## 2.7 Exponentiation

While no new operators are in ES2015, the exponentiation operator `**` has been proposed[2] for inclusion in the next version of the standard (at the time of this writing, ES2016), which makes the syntax `e1 ** e2` sugar for the built-in JavaScript function `Math.pow(e1, e2)`.

Sweet.js supports these kinds of syntax forms with *custom operators* [Rafkind and Flatt, 2012]. The custom operator definition of exponentiation is:

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

Operators have precedence and associativity so our custom binary operator definition set the precedence (14) and associativity (right associative) of exponentiation[3]. Precedence in sweet.js is represented by a number where bigger numbers bind more tightly. For example, by default addition (`+`) and subtraction (`-`) have a precedence of `12` whereas multiplication (`*`) and division (`/`) have a precedence of `13` and thus

[2]https://github.com/rwaldron/exponentiation-operator

[3]We need to surround the operator name `**` with parentheses because the JavaScript lexer considers `**` to be two separate `*` tokens. Surrounding a macro or operator name with parentheses in a declaration tells sweet.js to create a *multi-token* macro.

**Figure 2.4: Exponentiation Expansion**

```
y + x ** 10 ** 100 - z
```

expands to ⇒

```
y + Math.pow(x, Math.pow(10, 100)) - z;
```

bind more tightly.

Because custom operators allow us to specify the precedence and associativity the expression in Figure 2.4 expands correctly.

Custom operators in sweet.js are similar to macros in the sense that they match syntax and expand into new syntax, they differ in that they cannot match on arbitrary patterns. Rather, custom operators match only expressions that appear to the left and right of the operator (or just a single expression in the case of unary operators). This restriction allows sweet.js to properly handle precedence and associativity.

## 2.8 Bending Hygiene

Bending hygiene is done by stealing the lexical context from syntax objects in the "right place". To clarify, consider `aif` the anaphoric if macro that binds its condition to the identifier `it` in the body.

```
var it = "foo";
long.obj.path = [1, 2, 3];
aif (long.obj.path) {
    console.log(it);
}
// logs: [1, 2, 3]
```

This is a violation of hygiene because normally `it` should be bound to the surrounding environment (`"foo"` in the example above) but `aif` must capture `it`. To capture `it` inside `aif`, we can create an `it` binding in the macro that has the lexical context associated with the surrounding environment. The lexical context we want is actually found on the `aif` macro name itself. So we just need to create a new `it` binding using the lexical context of `aif`:

```
macro aif {
    case {
        // bind the macro name to `$aif_name`
        $aif_name
        ($cond ...) {$body ...}
    } => {
        // make a new `it` identifier using the lexical context
        // from `$aif_name`
        var it = makeIdent("it", #{$aif_name});
        letstx $it = [it];
        return #{
            // bind `$cond` to `$it`
            (function ($it) {
                if ($cond ...) {
                    // all `it` identifiers in `$body` will now
                    // be bound to `$it`
                    $body ...
                }
            })($cond ...);
        }
    }
}
```

The bending described in the section is classic Scheme-style hygiene bending. This technique has a number of limitations, especially when attempting to compose two macros that both bend hygiene. The Racket community has introduced an alternative technique called *syntax parameters* [Barzilay et al., 2011] that addresses a number of the problems with traditional hygiene bending.

## 2.9 Faithful Class

The `class` form we implemented earlier in this chapter was not fully faithful to the ES2015 standard because it was missing the `super` keyword that allows the constructor or a method access to the base class. We have now introduced enough sweet.js features to properly implement super.

```
class Bar extends Foo {
    constructor() {
        // refers to the base object's constructor
        super("yeah");
    }
    foo() {
        // refers to the base object's prototype
        super["baz"]
    }
    bar() {
        // refers to the base object's prototype
        super.baz;
    }
}
```

First off, we can separate out the method matching into a macroclass definition to make our main class macro more readable.

```
macroclass class_method {
    rule {
        $name ($args ...) {
            $body ...
        }
    }
}
```

The key to super is that it needs to refer to the prototype of the extending object. For example, in a constructor of a class Foo, super("foo") should be equivalent to:

```
Object.getPrototype(Foo.prototype)
      .constructor
      .call(this, "foo")
```

So, during expansion super needs to refer to its enclosing class name (Foo). A straightforward implementation technique here is to use some hygiene bending. We do this by defining a super macro whose name is hygienically bent to the same scope as the class definition (so all reference to `super` in the class definition are captured). Thus, each expansion of `class` includes a specialized declaration of `super` that knows the name of the enclosing class.

```
macro class {
    case {$cname
        $name extends $base {
            constructor ($params ...) { $body ... }

            $m:class_method ...
        }
    } => {
        letstx $super = [makeIdent("super", #{$cname})];
        return #{
            macro $super {
                rule { ($[$args ...]) } => {
                Object.getPrototypeOf($name.prototype)
                      .constructor
                      .call(this, $[$args ...])
                }
                rule { [$[$args ...]] } => {
                    Object.getPrototypeOf($name.prototype)[$[$args ...]]
                }
                rule { . $[$id] } => {
                    Object.getPrototypeOf($name.prototype)[to_str $[$id]]
                }
            }
            function $name($params ...) {
                $body ...
            }
            $name.prototype = Object.create($base.prototype);
            $(
            $name.prototype.$m$name = function($m$args ...) {
                $m$body ...
            }
            ) ...
        }
    }
}
```

We need a little utility macro `to_str` to help out with `super` that just converts an identifier syntax object (e.g., `foo`) to a string literal (λ eg,“foo”).

```
macro to_str {
    case {_ $x } => {
        letstx $str = [makeValue(unwrapSyntax(#{$x}), #{here})];
        return #{$str};
    }
}
```

## 2.10 Growing JavaScript

The macros shown in this section demonstrate one of the primary goals of sweet.js, namely giving programmers the ability to seamlessly grow their own language. While it is useful for JavaScript implementations to support these ES2015 language features, macros allow programmers to begin using these features without waiting for native support. Macros also enable grassroots language design whereby programmers can use macros to explore different syntax features and provide real-world feedback for future versions of JavaScript.

# Chapter 3

# Reading JavaScript

Parsers give structure to unstructured source code. In a system without macros this structuring 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 [Harbison and Steele, 1984], work over just the token stream. However, to implement truly expressive Scheme-like macros that can manipulate groups of unparsed tokens, it is useful 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 because of 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 fails to distinguish between a regular expression/divide operator, it can result in incorrectly matched delimiters.

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

A key novelty in sweet.js is the design and implementation of a reader that

**Figure 3.1: AST for Simplified JavaScript**

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

correctly distinguishes between regular expression literals and the divide operator for full ES5 JavaScript[1]. For clarity of presentation, this chapter describes the implementation of `read` for the subset of JavaScript shown in Figure 3.1. This subset is similar to the full ES5 grammar but with non-essential productions elided (for example, rather than handle each operator, this subset only includes `+` and `/`). This subset keeps the essential complexity of the full version of `read` while allowing the presentation of the technique to remain as clear as possible.

## 3.1 Read Formalism

For the formalism of given here, the `read` algorithm will take a *Lexeme* sequence. Lexemes are the output of a very simple lexer, which is not defined here. This lexer does not receive feedback from the parser like the ES5 lexer does, so it does not distinguish between regular expressions and the divide operator. Rather it simply lexes slashes into the ambiguous `/` lexeme. Lexemes also include keywords, puncutators, the (unmatched) delimiters, and variable identifiers.

[1]Our implementation also has initial support for code written in the upcoming ES2015 version of JavaScript.

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

The job of `read` is then to produce a correct *Token* sequence from a *Lexeme* sequence. Tokens include regular expression literals $\texttt{/}r\texttt{/}$, where `r` is a regular expression pattern (modeled as either a variable or one of the delimiters). This presentation simplifies regular expression bodies to just a variable and the individual delimiters, which captures the essential problems of parsing regular expressions. Tokens also include fully matched delimiters with nested token sequences $\underline{\texttt{(}T\texttt{)}}$ and $\underline{\texttt{\{}T\texttt{\}}}$ rather than individual delimiters (token delimiters with their nested token sequences are written with an $\underline{\text{underline}}$ to distinguish them from individual lexeme delimiters).

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

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

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

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

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

For clarity of presentation, line numbers are left implicit unless required, in which case we can use the notation $\texttt{\{}^{l}$ where $l$ is a line number.

A token sequence is written by separating elements with a dot so the source string "`foo(/)/)`" is lexed into a sequence of six lexemes `foo`·`(`·`/`·`)`·`/`·`)`. The equivalent token sequence is: `foo` · $\underline{\texttt{(·/)/·)}}$.

## 3.2 Read Algorithm

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

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` · `+` · `/` · `}` · `/`) 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 was a parenthesis (e.g., `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) /}/   // regex
```

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

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

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

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

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

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

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

$$\texttt{read} : \mathit{Lexeme}^* \rightarrow \mathit{Token}^* \rightarrow \mathit{Bool} \rightarrow \mathit{Token}^*$$

The implementation of `read` shown in Figure 3.3 uses two disjoint sets, *Divide-Prefix* and *RegexPrefix* defined in Figure 3.4, that determine if a slash should be either a divide or the start of a regular expression. These sets are parametrized by a boolean indicating if the prefix is inside an object literal or a block statement. The `read` function also includes an auxiliary function, `isExprPrefix` defined in Figure 3.2, that is 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.

Note that `read` is *not* defined over all inputs; the undefined inputs of `read` correspond to parse errors in the grammar. For example, the input `x = /x` is a parse error since the `/` must be a regular expression literal but it does not have a closing `/`.

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

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

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

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

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

**Figure 3.2: Helper Function for `read`**

$$\begin{array}{lcll}
\texttt{isExprPrefix} : \textit{Token}^* \to \textit{Bool} \to \textit{Int} \to \textit{Bool} & & & \\
\texttt{isExprPrefix}(\epsilon,\ \textit{true},\ l) & = & \textit{true} & \\
\texttt{isExprPrefix}(P \cdot \texttt{/},\ b,\ l) & = & \textit{true} & \\
\texttt{isExprPrefix}(P \cdot \texttt{+},\ b,\ l) & = & \textit{true} & \\
\texttt{isExprPrefix}(P \cdot \texttt{=},\ b,\ l) & = & \textit{true} & \\
\texttt{isExprPrefix}(P \cdot \texttt{:},\ b,\ l) & = & b & \\
\texttt{isExprPrefix}(P \cdot t \cdot \texttt{return}^l,\ b,\ l') & = & \textit{false} & \text{if } l \neq l' \text{ and } t \neq \texttt{.} \\
\texttt{isExprPrefix}(P \cdot t \cdot \texttt{return}^l,\ b,\ l') & = & \textit{true} & \text{if } l = l' \text{ or } t = \texttt{.} \\
\texttt{isExprPrefix}(P,\ b,\ l) & = & \textit{false} & \text{if other cases do not match}
\end{array}$$

## 3.3 Proving Read

To show that our `read` algorithm correctly distinguishes divide operations from regular expression literals, it is necessary to show that a parser defined over lexemes produces the same AST as a parser defined over the tokens generated by `read`.

The grammar for lexemes is defined in Figures 3.5 and 3.6, and generates ASTs in the abstract syntax shown in Figure 3.1. I use the notation $L = \textit{Program}_e$ to mean

**Figure 3.3: Read Algorithm**

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\texttt{read} : \mathit{Lexeme}^* \rightarrow \mathit{Token}^* \rightarrow \mathit{Bool} \rightarrow \mathit{Token}^*} \\
\texttt{read}(\texttt{/} \cdot L,\ P,\ b) & = & \texttt{/} \cdot \texttt{read}(L,\ P \cdot \texttt{/},\ b) \\
\quad \text{if } P \in \mathit{DividePrefix}_b & & \\
\texttt{read}(\texttt{/} \cdot r \cdot \texttt{/} \cdot L,\ P,\ b) & = & \texttt{/}r\texttt{/} \cdot \texttt{read}(L,\ P \cdot \texttt{/}r\texttt{/},\ b) \\
\quad \text{if } P \in \mathit{RegexPrefix}_b & & \\
\texttt{read}(\texttt{(} \cdot L \cdot \texttt{)} \cdot L',\ P,\ b) & = & \underline{\texttt{(}T\texttt{)}} \cdot \texttt{read}(L',\ P \cdot \underline{\texttt{(}T\texttt{)}},\ b) \\
\quad \text{if } L \text{ contains no unmatched } \texttt{)} & & \quad \text{if } T = \texttt{read}(L,\ \epsilon,\ \mathit{true}) \\
\texttt{read}(\texttt{\{}^l \cdot L \cdot \texttt{\}} \cdot L',\ P,\ b) & = & \underline{\texttt{\{}T\texttt{\}}}^l \cdot \texttt{read}(L',\ P \cdot \underline{\texttt{\{}T\texttt{\}}}^l,\ b) \\
\quad \text{if } L \text{ contains no unmatched } \texttt{\}} & & \\
\qquad T = \texttt{read}(L,\ \epsilon,\ \texttt{isExprPrefix}(P,\ b,\ l)) & & \\
\texttt{read}(k \cdot L,\ P,\ b) & = & k \cdot \texttt{read}(L,\ P \cdot k,\ b) \\
\quad \text{if } k \notin \{\texttt{/}, \texttt{(}, \texttt{\{}\} & & \\
\texttt{read}(\epsilon,\ P,\ b) & = & \epsilon
\end{array}
$$

**Figure 3.4: Read Prefix Helpers**

$$
\begin{array}{rcll}
\mathit{RegexPrefix}_b & ::= & \epsilon & \\
 & | & P \cdot t & \mathit{if}\ t \in \mathit{Punctuator} \\
 & | & P \cdot t \cdot t' & \mathit{if}\ t' \in \mathit{Keyword}\ \mathit{and}\ t \neq \texttt{.} \\
 & | & P \cdot t \cdot \texttt{if} \cdot \underline{\texttt{(}T\texttt{)}} & \mathit{if}\ t \neq \texttt{.} \\
 & | & P \cdot \texttt{function}^l \cdot x \cdot \underline{\texttt{(}T\texttt{)}} \cdot \underline{\texttt{\{}T'\texttt{\}}} & \mathit{if}\ \texttt{isExprPrefix}(P,\ b,\ l) = \mathit{false} \\
 & | & P \cdot \underline{\texttt{\{}T\texttt{\}}}^l & \mathit{if}\ \texttt{isExprPrefix}(P,\ b,\ l) = \mathit{false} \\
 & & & \\
\mathit{DividePrefix}_b & ::= & P \cdot x & \\
 & | & P \cdot \texttt{/}r\texttt{/} & \\
 & | & P \cdot t \cdot t' \cdot \underline{\texttt{(}T\texttt{)}} & \mathit{if}\ (t' \neq \texttt{if})\ \mathit{or}\ (t' = \texttt{if}\ \mathit{and}\ t = \texttt{.}) \\
 & | & P \cdot \underline{\texttt{\{}T\texttt{\}}}^l & \mathit{if}\ \texttt{isExprPrefix}(P,\ b,\ l) = \mathit{true}
\end{array}
$$

that $L$ is recognized by a derivation of *Program* producing the AST $e$.

Note that the grammar presented here is a simplified version of the grammar specified in the ECMAScript 5.1 standard [International, 2011] and many nonterminal names are shortened versions of nonterminals in the standard. It is mostly straightforward to extend the algorithm presented here to the full sweet.js implementation for ES5 JavaScript.

In addition to the *Program* parser just described, there also is a parser *Program′* that works over tokens. The rules of the two parsers are identical, except the rules with delimiters and regular expression literals change as follows:

$$\begin{array}{rcl}
PrimaryExpr_{\texttt{/}r\texttt{/}} & ::= & \texttt{/} \cdot r \cdot \texttt{/} \\
PrimaryExpr'_{\texttt{/}r\texttt{/}} & ::= & \texttt{/}r\texttt{/} \\
PrimaryExpr_{\texttt{(}e\texttt{)}} & ::= & \texttt{(} \cdot AssignExpr_e \cdot \texttt{)} \\
PrimaryExpr'_{\texttt{(}e\texttt{)}} & ::= & \underline{\texttt{(}AssignExpr'_e\texttt{)}}
\end{array}$$

To prove that `read` is correct, it is necessary to show that the following two parsing strategies give identical behavior:

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

**Theorem 1** (Parse Equivalence)**.**

*For all* $e \in AST$, $L \in Lexeme$:

*If* $L = Program_e$ *then* $\texttt{read}(L,\ \epsilon,\ \mathit{false}) = Program'_e$

*Proof.* The proof proceeds by induction on the derivation of $Program_e$ to show that parse equivalence holds between all corresponding nonterminals in the two grammars. In particular, parse equivalence holds by Lemma 1 shown in the Appendix. □

**Figure 3.5: Simplified ES5 Grammar Pt. 1**

$$
\begin{array}{lcl}
\mathit{PrimaryExpr}_{x} & ::= & x \\
\mathit{PrimaryExpr}_{\texttt{/}r\texttt{/}} & ::= & \texttt{/} \cdot r \cdot \texttt{/} \\
\mathit{PrimaryExpr}_{\texttt{\{}x\texttt{:}e\texttt{\}}} & ::= & \texttt{\{} \cdot x \cdot \texttt{:} \cdot \mathit{AssignExpr}_{e} \cdot \texttt{\}} \\
\mathit{PrimaryExpr}_{\texttt{(}e\texttt{)}} & ::= & \texttt{(} \cdot \mathit{AssignExpr}_{e} \cdot \texttt{)} \\
 & & \\
\mathit{MemberExpr}_{e} & ::= & \mathit{PrimaryExpr}_{e} \\
\mathit{MemberExpr}_{e} & ::= & \mathit{Function}_{e} \\
\mathit{MemberExpr}_{e\texttt{.}x} & ::= & \mathit{MemberExpr}_{e} \cdot \texttt{.} \cdot x \\
\mathit{CallExpr}_{e\texttt{(}e'\texttt{)}} & ::= & \mathit{MemberExpr}_{e} \cdot \texttt{(} \cdot \mathit{AssignExpr}_{e'} \cdot \texttt{)} \\
\mathit{CallExpr}_{e\texttt{(}e'\texttt{)}} & ::= & \mathit{CallExpr}_{e} \cdot \texttt{(} \cdot \mathit{AssignExpr}_{e'} \cdot \texttt{)} \\
\mathit{CallExpr}_{e\texttt{.}x} & ::= & \mathit{CallExpr}_{e}\ \texttt{.}\ x \\
 & & \\
\mathit{LHSExpr}_{e} & ::= & \mathit{MemberExpr}_{e} \\
\mathit{LHSExpr}_{e} & ::= & \mathit{CallExpr}_{e} \\
 & & \\
\mathit{BinaryExpr}_{e} & ::= & \mathit{LHSExpr}_{e} \\
\mathit{BinaryExpr}_{e\ \texttt{/}\ e'} & ::= & \mathit{BinaryExpr}_{e} \cdot \texttt{/} \cdot \mathit{BinaryExpr}_{e'} \\
\mathit{BinaryExpr}_{e\ \texttt{+}\ e'} & ::= & \mathit{BinaryExpr}_{e} \cdot \texttt{+} \cdot \mathit{BinaryExpr}_{e'} \\
\mathit{AssignExpr}_{e} & ::= & \mathit{BinaryExpr}_{e} \\
\mathit{AssignExpr}_{e\ \texttt{=}\ e'} & ::= & \mathit{LHSExpr}_{e} \cdot \texttt{=} \cdot \mathit{AssignExpr}_{e'}
\end{array}
$$

**Figure 3.6: Simplified ES5 Grammar Pt. 2**

| | | |
|---|---|---|
| $\mathit{StmtList}_{e}$ | $::=$ | $\mathit{Stmt}_{e}$ |
| $\mathit{StmtList}_{e\ e'}$ | $::=$ | $\mathit{StmtList}_{e} \cdot \mathit{Stmt}_{e'}$ |
| $\mathit{Stmt}_{\texttt{\{}e\texttt{\}}}$ | $::=$ | $\texttt{\{} \cdot \mathit{StmtList}_{e} \cdot \texttt{\}}$ |
| $\mathit{Stmt}_{\texttt{x:}\ e}$ | $::=$ | $\texttt{x} \cdot \texttt{:} \cdot \mathit{Stmt}_{e}$ |
| $\mathit{Stmt}_{e}$ | $::=$ | $\mathit{AssignExpr}_{e} \cdot \texttt{;} \qquad \textit{where lookahead} \neq \texttt{\{}\ \textit{or}\ \texttt{function}$ |
| $\mathit{Stmt}_{\texttt{if (}e\texttt{)}\ e'}$ | $::=$ | $\texttt{if} \cdot \texttt{(} \cdot \mathit{AssignExpr}_{e} \cdot \texttt{)} \cdot \mathit{Stmt}_{e'}$ |
| $\mathit{Stmt}_{\texttt{return}}$ | $::=$ | $\texttt{return}$ |
| $\mathit{Stmt}_{\texttt{return}\ e}$ | $::=$ | $\texttt{return} \cdot \text{[no line terminator here]}\ \mathit{AssignExpr}_{e} \cdot \texttt{;}$ |
| $\mathit{Function}_{\texttt{function}\ x\ \texttt{(}x'\texttt{)}\ \texttt{\{}e\texttt{\}}}$ | $::=$ | $\texttt{function} \cdot x \cdot \texttt{(} \cdot x' \cdot \texttt{)} \cdot \texttt{\{} \cdot \mathit{SourceElements}_{e} \cdot \texttt{\}}$ |
| $\mathit{SourceElement}_{e}$ | $::=$ | $\mathit{Stmt}_{e}$ |
| $\mathit{SourceElement}_{e}$ | $::=$ | $\mathit{Function}_{e}$ |
| $\mathit{SourceElements}_{e}$ | $::=$ | $\mathit{SourceElement}_{e}$ |
| $\mathit{SourceElements}_{e\ e'}$ | $::=$ | $\mathit{SourceElements}_{e} \cdot \mathit{SourceElement}_{e'}$ |
| $\mathit{Program}_{e}$ | $::=$ | $\mathit{SourceElements}_{e}$ |

# Chapter 4

# Expansion in JavaScript

In Scheme, `expand` is the core function of the macro system responsible for taking the s-expressions generated by `read` and discovering, loading, and expanding any macros in the code. The Scheme compiler pipeline looks something like:

$$\texttt{read} \xrightarrow{Sexp} \texttt{expand} \xrightarrow{Sexp} \texttt{parse} \xrightarrow{AST} \texttt{eval}$$

In a language with s-expressions like Scheme, this pipeline is sufficient to implement very expressive macro systems [Matthew Flatt and PLT, 2010]. Fully delimited s-expressions play a key role in enabling the expressive power of this pipeline.

However in a language like JavaScript, since most of the syntax forms 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.

To clarify, consider the expansion of simple `let` macro in Figure 4.1. Like many syntactic forms in JavaScript, the initialization expression of the let declaration is an undelimited sequence of tokens. An expressive macro system necessitates macros that can match and manipulate patterns such as an expression. In sweet.js, the ability to match on these undelimited sequences is provided via the `:expr` pattern.

Sweet.js groups tokens by transforming a token tree into a *term* through a technique pioneered by the Honu language called *enforestation* [Rafkind, 2013, Rafkind and Flatt, 2012]. Enforestation works by progressively recognizing and grouping (potentially undelimited) syntax forms (e.g., literals, identifiers, expressions, and statements) during expansion. Essentially, enforestation delimits undelimited syntax.

**Figure 4.1: Let Expansion**

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

expands to ⇒

```
var x = 40 + 2;
```

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

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 making a first pass through the top-level scope that loads the `let` macro into the macro environment and converts the function declaration into a term (here I use angle brackets to denote the *term* data structure). Notice that the body of the function is kept unenforested.

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

Next, a term 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 descends into the function body. The use of the `let` macro is expanded and the `var` and `return` term are created.

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

## 4.1 Expansion with Infix

In a traditional macros system like Scheme the macro name occurs before the syntax it matches. For example, if `m` has been defined as a macro then an invocation like `(m foo bar baz)` the `m` macro transformer will be invoked with the entire s-expression `(m foo bar baz)`. A macro transformer in Scheme thus has something like the type $Sexp \rightarrow Sexp$.

Sweet.js diverges from the Scheme approach in two ways. First, since there are no s-expressions that delimit the extent of a macro invocation, macros are invoked with all of the tokens following the macro name; a macro can match and manipulate as many or as few tokens as it wants, up to the next closing delimiter. Second, macros are also invoked with all of the tokens preceding the macro name (again up to the nearest delimiter); providing the preceding tokens to the syntax transformer allows us to implement infix macros.

So the most obvious type for a transformer in sweet.js would be:

$$(\mathit{Token}^*, \mathit{Token}^*) \rightarrow (\mathit{Token}^*, \mathit{Token}^*)$$

Here, the transformer is given two arguments, the first for the sequence of tokens that precede the macro name and the second with the sequence of tokens that follow. The transformer could then consume from either end as needed, yielding new preceding and following tokens.

However, 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.

The key problem here is that the term structure already discovered by the expander is violated by matching only part of the tokens contained within the term. To address this problem the macro actually takes the *terms* previously enforested rather than *tokens*. So the actual type is:

$$(\mathit{Term}^*, \mathit{Token}^*) \rightarrow (\mathit{Term}^*, \mathit{Token}^*)$$

The transformer can match on and modify both the prefix of terms and the tokens following the macro identifier.

So now in our running example:

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

is first enforested to:

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

So now the `=>` transformer just sees a call term in its prefix argument and correctly fails the match.

## 4.2 Expansion Formalized

To formalize these intuitions, I now consider a small JavaScript-like language that contains just the bare minimum to demonstrate how expansion and enforestation works. The grammar of this language is as follows:

$$\begin{array}{lcl}
e & ::= & \mathit{stmt} \mid \mathit{expr} \\
\mathit{stmt} & ::= & \texttt{var}\ x = \mathit{expr} \mid \texttt{if}\ (\mathit{expr})\ \{e^*\} \\
\mathit{expr} & ::= & x \mid n \mid \mathit{expr}\ \mathit{op}\ \mathit{expr} \mid \mathit{expr}\ (\mathit{expr}) \mid \texttt{function}\ x\ (x)\ \{e^*\} \\
\mathit{op} & ::= & + \mid - \mid * \mid / \mid \ldots \\
AST & ::= & e^*
\end{array}$$

Like JavaScript, our language distinguishes between statements and expressions. Statements include variable declarations ($\texttt{var}\ x = \mathit{expr}$) and conditional statements ($\texttt{if}\ (\mathit{expr})\ \{e^*\}$)[1]. Expressions can be variables $x$, numbers $n$, binary operators $\mathit{expr}\ \mathit{op}\ \mathit{expr}$, function calls $\mathit{expr}\ (\mathit{expr})$, and function declarations $\texttt{function}\ x\ (x)\ \{e^*\}$. For simplicity, functions only accept a single argument.

The AST of a program is then a sequence of statements and expressions. Also like JavaScript, our language does not require semicolons to terminate lines; in fact, our simplified model does not include any semicolons at all so the following example would be recognized as a variable declaration followed by a binary expression:

```
var x = 2 + 2
x * 100
```

While this language is not terribly useful to program in, it does demonstrate the key syntactic forms that make expansion complex. In particular, it includes flat

[1]missing from our simplified language are some JavaScript features such as loops and switch statements

**Figure 4.2: Grammar for Enforest**

| | | | |
|---|---|---|---|
| $x,\ name$ | $\in$ | $Variable$ | **Variable** |
| $t \in Token$ | $::=$ | $x \mid n \mid bop \mid \underline{\texttt{(}S\texttt{)}} \mid \underline{\texttt{\{}S\texttt{\}}}$ | **Token Tree** |
| $s \in Syntax$ | $::=$ | $t^C$ | **Syntax Object** |
| $id \in Identifier$ | $::=$ | $x^C$ | **Identifiers** |
| $C \in Scopeset$ | $::=$ | $\emptyset$ | **Scopeset** |
| | | | |
| $m$ | $::=$ | $ExprTerm \mid StmtTerm$ | **Term** |
| $ExprTerm$ | $::=$ | $id$ | Identifier |
| | | $n$ | Numeric Literal |
| | | $m\ op\ m$ | Binary Operator |
| | | $m\ \underline{\texttt{(}S\texttt{)}}$ | Call |
| | | $\texttt{function}\ id\ \underline{\texttt{(}S\texttt{)}}\ \underline{\texttt{\{}S\texttt{\}}}$ | Function Declaration |
| $StmtTerm$ | $::=$ | $\texttt{var}\ id\ \texttt{=}\ m$ | Variable Declaration |
| | | $\texttt{if}\ \underline{\texttt{(}S\texttt{)}}\ \underline{\texttt{\{}S\texttt{\}}}$ | If Statement |
| | | $\texttt{macro}\ id\ \underline{\texttt{\{}S\texttt{\}}}$ | Macro Declaration |
| | | | |
| $S$ | $::=$ | $s^*$ | **Syntax Sequence** |
| $M$ | $::=$ | $m^*$ | **Term Sequence** |
| $a$ | $::=$ | $left \mid right$ | **Associativity** |
| $binary$ | $::=$ | $+ \mid - \mid * \mid / \mid \ldots$ | |
| $bop$ | $::=$ | $binary^{a,n}$ | |

**Figure 4.3: Expansion Environment**

$$\begin{array}{rcl}
E \in \mathit{Env} & ::= & \mathit{Name} \rightarrow_p \mathit{Transform} \\
\\
\mathit{Transform} & ::= & \mathit{PrimTrans} \mid \mathit{MacroTrans} \\
\mathit{PrimTrans} & ::= & \textbf{VarDecl} \mid \textbf{FunDecl} \mid \textbf{MacroDecl} \mid \textbf{IfStmt} \\
\mathit{MacroTrans} & ::= & (\mathit{Match}, \mathit{Trans}) \\
\\
\mathit{Match} & ::= & (\mathit{Term}^*, \mathit{Syntax}^*) \rightarrow (\mathit{Subst}, \mathit{Term}^*, \mathit{Syntax}^*) \\
\mathit{Trans} & ::= & \mathit{Subst} \rightarrow \mathit{Syntax}^* \\
\\
\mathit{Subst} & ::= & \mathit{Variable} \rightarrow_p \mathit{Syntax}^*
\end{array}$$

binding forms (variable declarations), nested binding forms (function declarations), and operators.

The data types used for expansion, shown in Figure 4.2, are tokens ($t$), syntax objects ($s$, a sequence of which is written $S$), and terms ($m$, a sequence of which is written $M$).

Tokens are similar to what we saw in Chapter 3 and include variabls $x$, numeric literals $n$, binary operators *bop*, and two delimiter tokens with subtrees, parentheses $\underline{(S)}$ and braces $\underline{\{S\}}$, that nest a syntax sequence (square brackets are omitted here since this minimal language does not include array literals or other syntax forms that would require them).

Syntax objects are tokens with a scopeset ($C$). Scopesets are used to keep track of binding information for the hygiene algorithm. This chapter describes a non-hygienic version of expansion and so all scopesets are just the empty set. Chapter 5 expands this formalism with hygiene.

In other contexts the words "variable", "name", and "identifier" are roughly interchangeable. Here, these concepts are kept separate. A *variable* ($x$) and a *name* (*name*) are both elements of the same set and represent tokens such as `foo` or `myFunctionName42`, however variables appear in the source program while names are used to represent bindings for hygiene. Since this chapter does not address hygiene yet, no fresh names are generated in the formalism presented here; Chapter 5 will introduce a distinction between variables and names. An *identifier* is a variable syntax object ($x^C$).

Terms are a kind of proto-AST in the sense that while terms contain the same productions as the AST defined earlier, terms can contain unexpanded syntax sequences as their subtrees. In this way a term represents a partially parsed program. This partial parse is necessary because expansion requires multiple passes through a given scope; the first pass will create terms for each form in the scope while the second pass recurses into sub-terms.

Simplifying a little bit, the process of expanding macros will look something like this:

$$\texttt{read} \xrightarrow{\textit{Syntax}^*} \texttt{expand} \xrightarrow{\textit{Term}^*} \texttt{parse} \xrightarrow{\textit{AST}} \texttt{eval}$$

While this gets intuitively at what is going on, `expand` is broken up into a multi-step process since we need to perform multiple passes:

$$\xrightarrow{\textit{Syntax}^*} \texttt{expandTokens} \xrightarrow{\textit{Term}^*} \texttt{expandTerms} \xrightarrow{\textit{Term}^*}$$

The high-level intuition for these functions is that `expandTokens` is responsible for recognizing the basic term forms (via *enforestation*) and installing macro definitions into the compile-time environment while `expandTerms` is responsible for traversing into nested term structures and dealing with some hygiene issues (e.g., the parameter bindings in functions). Basically, `expandTokens` performs the first pass through a scope and `expandTerms` performs the second pass.

Two passes allows macro invocations to appear in nested structures prior to their (lexical) declarations. For example, in the following snippet the `id` macro appears inside the `foo` function before its definition.

```
function foo() {
    id 42
}
```

```
macro id {
    rule { $x } => { $x }
}
```

In the first pass over this code, `expandTokens` will load the macro definition into the compile-time environment without traversing into the function body of `foo`. On the second pass `expandTerms` can traverse into `foo` and expand the `id` macro.

Expansion works over an environment, defined in Figure 4.3, that maps names to transforms. A transform represents the meaning of a binding and is either a primitive transform such as `FunDecl` (meaning the binding should be interpreted as the built-in function form) or a macro transform. For the purposes of this formalism, a macro transform is modeled as a pair of *match* and *trans* functions that perform pattern matching and transcription respectively. I describe macro transformers more in section 4.3.

The environment is initialized with the built-in keywords mapped to their respective primitive transforms (e.g., $E(\texttt{var}) = \texttt{VarDecl}$), which may seem excessive but will become important when hygiene is introduced in Chapter 5.

Expansion uses the `resolve` function defined in Figure 4.5 to determine the name an identifier is bound to. When hygiene is introduced in Chapter 5 `resolve` will be a key function, but here it simply strips the scopeset from an identifier.

## 4.3 Expanding Tokens

Expansion begins, appropriately enough, with the `expand` function (Figure 4.4), which simply stitches together `expandTokens` and `expandTerms`. The `expandTokens` function works over a prefix of previously encountered terms (retaining this prefix is a divergence from other macro expansion techniques and allows us to implement infix macros) along with the remaining syntax objects to process and a compile-time environment (which stores macro definitions). The `expand` function calls `expandTokens` with an empty prefix and environment along with the tokens tree sequence to expand. The resulting pair of terms and the modified environment are then fed into `expandTerms`.

The key idea of `expandTokens` is that it uses $\rightarrow^*_{E,n}$, the reflexive transitive closure of the enforest relation $\rightarrow_{E,n}$ defined in Figure 4.6, to enforest the next term in

**Figure 4.4: Expand Function**

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\texttt{expand} : \textit{Syntax}^* \rightarrow \textit{Term}^*} \\
\texttt{expand}(S) & = & \text{let } (M, E) = \texttt{expandTokens}(\epsilon, S, \emptyset) \\
 & & \text{in } \ \texttt{expandTerms}(M, E) \\
 & & \\
\multicolumn{3}{l}{\texttt{expandTokens} : (\textit{Term}^*, \textit{Syntax}^*, \textit{Env}) \rightarrow (\textit{Term}^*, \textit{Env})} \\
\texttt{expandTokens}(M, \epsilon, E) & = & (\texttt{reverse}(M), E) \\
\texttt{expandTokens}(M, S, E) & = & \texttt{expandTokens}(\texttt{macro}\ \textit{id}\ \underline{\texttt{\{}S_1\texttt{\}}} \cdot M', S', E') \\
\multicolumn{3}{l}{\quad \text{if } M,\ \bot,\ S \rightarrow^*_{E,0} M',\ \texttt{macro}\ \textit{id}\ \underline{\texttt{\{}S_1\texttt{\}}},\ S'} \\
\multicolumn{3}{l}{\qquad E' = E[\textit{id} := \texttt{loadMacro}(S_1)]} \\
\texttt{expandTokens}(M, S, E) & = & \texttt{expandTokens}(m \cdot M', S', E) \\
\multicolumn{3}{l}{\quad \text{if } M,\ \bot,\ S \rightarrow^*_{E,0} M',\ m,\ S'}
\end{array}
$$

the token sequence. Enforestation works by building the longest possible term from the tokens. For example, running enforestation over the tokens $2 \cdot \texttt{+} \cdot 2 \cdot 10 \cdot \texttt{-} \cdot 5$ will produce the term $2 + 2$ followed by the remaining tokens $10 \cdot \texttt{-} \cdot 5$.

If the enforested term is a macro definition, then it is added to the environment. Here loading of a macro is modeled by the primitive `loadMacro` that returns a pair of a matcher function and a transformer function (*match*, *trans*). The matcher function models pattern matching by taking the prefix terms and remaining syntax objects and returns a substitution environment $\textit{match} : (\textit{Term}^*, \textit{Token}^*) \rightarrow \textit{Subst}$. A substitution environment maps pattern variables to a sequence of syntax objects for use in the transformer which takes the substitution environment and returns a syntax object sequence $\textit{trans} : \textit{Subst} \rightarrow \textit{Token}^*$.

In the implementation of sweet.js, a macro definition is actually compiled to a JavaScript function. For the purposes of formalization considered here, the `loadMacro` primitive is sufficient to capture the essential complexity of macro definitions, namely manipulating sequences of terms and tokens.

**Figure 4.5: Resolve Function**

$$\begin{array}{lcl} \texttt{resolve} : \mathit{Identifier} \rightarrow \mathit{Name} & & \\ \texttt{resolve}(x^C) & = & x \end{array}$$

Once enforested the term is added to the prefix and expansion continues. Once all of the tokens are enforested, the prefix is reversed and returned along with the environment.

## 4.4 Enforestation

Enforestation is the key piece of expansion that both performs macro expansion and delimits implicitly delimited syntax forms. Each step of enforestation is defined by the relation $\rightarrow_{E,n}$ (see Figure 4.6), which relates a triple of the already expanded terms (for use in infix macros), the term that enforest is currently working on (or $\perp$ meaning that the term has not been started yet), and the remaining syntax that has not been enforested yet. The relation is parameterized by $E$, the compile-time environment, and a number $n$, that represents the current operator precedence level.

$$\mathit{Term}_\perp = \{\perp\} \cup \mathit{Term}$$

$$(\rightarrow_{E,n}) \subseteq (\mathit{Term}^*, \mathit{Term}_\perp, \mathit{Syntax}^*) \times (\mathit{Term}^*, \mathit{Term}_\perp, \mathit{Syntax}^*)$$

The basic idea of enforestation is to consume as many syntax objects from the syntax sequence as possible to construct a term.

Beginning from $\perp$, if the first syntax object in the syntax sequence is an identifier or a numeric literal, enforest steps to an identifier or numeric literal term respectively.

In the cases where enforest has stepped to an identifier, the relation must check the *meaning* of each binding in the compile-time environment $E$. For example, the rule that handles function declaration terms checks that the identifier $\mathit{id}_{\mathit{fun}}$ is bound

**Figure 4.6: Enforest Function**

$$
\begin{array}{rcl}
M,\ \bot,\ \mathit{id} \cdot S & \rightarrow_{E,n} & M,\ \mathit{id},\ S \\
M,\ \bot,\ n_1^C \cdot S & \rightarrow_{E,n} & M,\ n_1,\ S \\
M,\ \mathit{id},\ S & \rightarrow_{E,n} & M',\ \bot,\ S_{\mathit{res}} \cdot S' \\
 & \multicolumn{2}{r}{\text{if } (\mathit{match}, \mathit{trans}) = E(\texttt{resolve}(\mathit{id}))} \\
 & \multicolumn{2}{r}{(\theta, M', S') = \mathit{match}(M, S)} \\
 & \multicolumn{2}{r}{S_{\mathit{res}} = \mathit{trans}(\theta)} \\
M,\ \mathit{id}_{\mathit{var}},\ \mathit{id} \cdot \texttt{=} \cdot S & \rightarrow_{E,n} & M,\ \texttt{var}\ \mathit{id}\ m,\ S' \\
 & \multicolumn{2}{r}{\text{if } \textbf{VarDecl} = E(\texttt{resolve}(\mathit{id}_{\mathit{var}}))} \\
 & \multicolumn{2}{r}{\epsilon,\ \bot,\ S \rightarrow^{*}_{E,0} \epsilon,\ m,\ S'} \\
 & \multicolumn{2}{r}{m \in \mathit{ExprTerm}} \\
M,\ \mathit{id}_{\mathit{fun}},\ \mathit{id}_1 \cdot \underline{\texttt{(}\mathit{id}_2\texttt{)}} \cdot \underline{\texttt{\{}S_1\texttt{\}}} \cdot S_2 & \rightarrow_{E,n} & M,\ \texttt{function}\ \mathit{id}_1\ \underline{\texttt{(}\mathit{id}_2\texttt{)}}\ \underline{\texttt{\{}S_1\texttt{\}}},\ S_2 \\
 & \multicolumn{2}{r}{\text{if } \textbf{FunDecl} = E(\texttt{resolve}(\mathit{id}_{\mathit{fun}}))} \\
M,\ \mathit{id}_{\mathit{if}},\ \underline{\texttt{(}S_1\texttt{)}} \cdot \underline{\texttt{\{}S_2\texttt{\}}} \cdot S_3 & \rightarrow_{E,n} & M,\ \texttt{if}\ \underline{\texttt{(}S_1\texttt{)}}\ \underline{\texttt{\{}S_2\texttt{\}}},\ S_3 \\
 & \multicolumn{2}{r}{\text{if } \textbf{IfStmt} = E(\texttt{resolve}(\mathit{id}_{\mathit{if}}))} \\
M,\ \mathit{id}_{\mathit{mac}},\ \mathit{id} \cdot \underline{\texttt{\{}S_1\texttt{\}}} \cdot S_2 & \rightarrow_{E,n} & M,\ \texttt{macro}\ \mathit{id}\ \underline{\texttt{\{}S_1\texttt{\}}},\ S_2 \\
 & \multicolumn{2}{r}{\text{if } \textbf{MacroDecl} = E(\texttt{resolve}(\mathit{id}_{\mathit{mac}}))} \\
M,\ m,\ \underline{\texttt{(}S\texttt{)}} \cdot S' & \rightarrow_{E,n} & M,\ m\ \underline{\texttt{(}S\texttt{)}},\ S' \\
 & \multicolumn{2}{r}{\text{if } m \in \mathit{ExprTerm}} \\
M,\ m_1,\ \mathit{bop}^{a,n_{op}} \cdot S & \rightarrow_{E,n} & M,\ m_1\ \mathit{bop}\ m_2,\ S' \\
 & \multicolumn{2}{r}{\text{if } n_{op} >_a n} \\
 & \multicolumn{2}{r}{\epsilon,\ \bot,\ S \rightarrow^{*}_{E,n_{op}} \epsilon,\ m_2,\ S'} \\
 & \multicolumn{2}{r}{m_1 \in \mathit{ExprTerm}} \\
 & \multicolumn{2}{r}{m_2 \in \mathit{ExprTerm}}
\end{array}
$$

to **FunDecl** transform. The rules for function declarations, if statements, and macro declarations are all similar since each simply step to their respective term after checking the compile-time environment for their primitive transforms. Note that their bodies are left as syntax sequences. This allows `expandTokens` to first discover and load all macro definitions in a given scope.

If the current term is an identifier that maps to a macro transform in the environment, then the macro is invoked by pulling the $(\mathit{match}, \mathit{trans})$ pair out of the environment. The prefix terms $M$ and remaining syntax objects $S$ are passed to *match* resulting in a substitution $\theta$, potentially modified prefix $M'$, and the remaining syntax sequence $S'$. Then *trans* is invoked with the substitution resulting in a syntax sequence $S_{res}$.

Variable declarations work by enforesting the initializer. The syntax sequence following the `=` token is enforested under an empty prefix (effectively the initializer is surrounded by an implicit delimiter). The initializer term must be an expression so, for example, `var x = var y = 42` would get stuck since `var y = 42` is a statement.

There are two cases to consider when a term is followed by a binary operator. The first case is where the precedence of the binary operator $n_{op}$ is greater than the current precedence $n$ of the enforestation context. In this case, the syntax to the right of the operator is run through the enforestation relation at the operator's precedence level $n_{op}$ until a new expression $m_2$ is built.

Associativity is handled by dispatching to the appropriately parameterized ordering operator:

$$\begin{aligned} n_1 <_{\mathit{left}} n_2 &\overset{\text{def}}{=} n_1 \leq n_2 \\ n_1 <_{\mathit{right}} n_2 &\overset{\text{def}}{=} n_1 < n_2 \end{aligned}$$

The other case, where the operator's precedence level is less than the context's precedence, is implicitly handled by enforestation. A lower operator precedence level simply means that enforestation at the current precedence level is finished and enforest can stop stepping.

For example, consider the enforestation steps of `1 * 2 + 3`:

$$\epsilon,\ \bot,\ 1 \cdot \texttt{*}^{\mathit{left},13} \cdot 2 \cdot \texttt{+}^{\mathit{left},12} \cdot 3$$
$$\rightarrow_{\emptyset,0}$$
$$\epsilon,\ 1,\ \texttt{*}^{\mathit{left},13} \cdot 2 \cdot \texttt{+}^{\mathit{left},12} \cdot 3$$

Since the precedence level of `*` (13) is greater than the context precedence (0), enforestation is run over the remaining syntax giving the following steps:

$$\epsilon,\ \bot,\ 2 \cdot \texttt{+}^{\mathit{left},12} \cdot 3$$
$$\rightarrow_{\emptyset,13}$$
$$\epsilon,\ 2,\ \texttt{+}^{\mathit{left},12} \cdot 3$$

Since the context's precedence (13) is greater than the precedence on `+` (12), there are no more steps to take and we return to the original trace.

$$\ldots$$
$$\rightarrow_{\emptyset,0}$$
$$\epsilon,\ 1{*}2,\ \texttt{+}^{\mathit{left},12} \cdot 3$$
$$\rightarrow_{\emptyset,0}$$
$$\epsilon,\ (1{*}2){+}3,\ \epsilon$$

Note that parentheses around `1 * 2` denote that `1 * 2` is a sub-term of `(1 * 2) + 3` but no "real" parentheses are present.

## 4.5 Expanding Terms

After `expandTokens` and the enforestation relation have completed a first pass through a given scope, `expandTerms` (Figure 4.7) is invoked on the partially expanded terms. In the formalization presented here the primary purpose of `expandTerms` is to invoke `expand` on the unexpanded tokens inside of function bodies and if branches. Deferring expansion of these tokens to a second pass enables all macro definitions in the scope to be discovered and loaded. As mentioned before, this allows the identity macro to be defined *after* its (textual) use:

```
function foo() {
    id 42
```

**Figure 4.7: expandTerms Function**

$$\texttt{expandTerms} : (\mathit{Term}^*, \mathit{Env}) \rightarrow \mathit{Term}^*$$

$$
\begin{array}{lcl}
\texttt{expandTerms}(\mathit{id} \cdot M, E) & = & \mathit{id} \cdot \texttt{expandTerms}(M, E) \\
\texttt{expandTerms}(n \cdot M, E) & = & n \cdot \texttt{expandTerms}(M, E) \\
\texttt{expandTerms}(m_1 \ \mathit{op} \ m_2 \cdot M, E) & = & m_1 \ \mathit{op} \ m_2 \cdot \texttt{expandTerms}(M, E) \\
\texttt{expandTerms}(\texttt{var} \ \mathit{id} \ \texttt{=} \ m \cdot M, E) & = & \texttt{var} \ \mathit{id} \ \texttt{=} \ m \cdot \texttt{expandTerms}(M, E) \\
\texttt{expandTerms}(m \ \underline{\texttt{(}S\texttt{)}} \cdot M, E) & = & \text{let } M_1 = \texttt{expand}(S, E) \\
 & & \text{in } \ m \ \underline{\texttt{(}M_1\texttt{)}} \cdot \texttt{expandTerms}(M, E) \\
\texttt{expandTerms}(\texttt{function} \ \mathit{id}_1 \ \underline{\texttt{(}\mathit{id}_2\texttt{)}} \ \underline{\texttt{\{}S_1\texttt{\}}} \cdot M, E) & = & \text{let } M_1 = \texttt{expand}(S_1, E) \\
 & & \quad M_2 = \texttt{expandTerms}(M, E) \\
 & & \text{in } \ \texttt{function} \ \mathit{id}_1 \ \underline{\texttt{(}\mathit{id}_2\texttt{)}} \ \underline{\texttt{\{}M_1\texttt{\}}} \cdot M_2 \\
\texttt{expandTerms}(\texttt{if} \ \underline{\texttt{(}S_1\texttt{)}} \ \underline{\texttt{\{}S_2\texttt{\}}} \cdot M, E) & = & \text{let } M_1 = \texttt{expand}(S_1, E) \\
 & & \quad M_2 = \texttt{expand}(S_2, E) \\
 & & \quad M_3 = \texttt{expandTerms}(M, E) \\
 & & \text{in } \ \underline{\texttt{(}M_1\texttt{)}} \ \underline{\texttt{\{}M_2\texttt{\}}} \cdot M_3
\end{array}
$$

```
}
macro id {
    rule { $x } => { $x }
}
```

The `expandTerms` function is also the location where some important hygiene operations take place. In particular, when `expandTerms` encounters a function declaration, the parameters to the function must be renamed in its body. This is described in Chapter 5.

## 4.6 Infix and Delimiters

Two important characteristics of the infix macro design are (1) that infix macros match a prefix of *terms* rather than syntax and (2) they cannot match outside of their containing delimiter.

While standard macros match on a sequence of tokens, the prefix for an infix macro is terms. This choice prevents infix macros from attempting to "split" terms during a match. For example, consider this use of the ES2015 arrow function:

```
macro => {
    rule infix { ($param) | $body:expr } => {
        function ($param) { return $body; }
    }
}
foo (x) => x
```

By only providing an infix macro with the previously enforested terms as a prefix, the `=>` macro will only have the single term `foo(x)` in the prefix sequence and thus fail to match.

In addition, infix macros cannot match outside of their containing delimiter. For example, the prefix available to `inf` in `foo(42 inf)` is just `42`; this macro cannot see or match against the identifier `foo`.

This behavior should not be too surprising since standard macros also are not able to match outside of their containing delimiter. However, unlike standard macros, infix macro are also trapped inside the *implicit* delimiters of operators. For example, consider an infix macro that attempts to match an addition operator and change it to subtraction:

```
macro inf {
    rule infix { + | } => { - }
}
42 + inf 42
```

This example will fail to match since in the enforestation rule that handles binary operators, the sub-enforestation of the right-hand side runs with the empty prefix. Essentially, enforestation adds implicit parentheses to binary operators.

Variable declarations are also enforested with an empty prefix. This empty prefix means that the following macro will fail:

```
let async = macro {
    rule infix { var $id = | $call:expr
                 $rest ...} => {
        $call.then(function($id) {
            $rest ...
        });
    }
}
var x = async f(42);
```

While the above example may seem reasonable at first glance, allowing infix macros to match before the initializer leads to complicated interactions with hygiene. In the above example, `x` starts as a variable binding so it should be bound in the surrounding scope; however, the infix macro moves `x` to a function parameter, which should only be bound in the function's scope. It is unclear what the hygiene algorithm should do in a case like this so I have chosen to simply disallow infix macros matching outside of the initializer expression.

In any event, the `async` behavior shown here could be equivalently implemented in a more reasonable way by creating a macro for `var` instead:

```
macro var {
    rule infix { $id = async $call:expr
                 $rest ...} => {
        $call.then(function($id) {
            $rest ...
        });
    }
}
var x = async f(42);
```

This implementation of `async` makes it clearer that the behavior of variable declarations is being changed and avoids any dubious interactions with bindings.

The implementation of sweet.js actually loosens the implicit delimiter restriction around infix and operators (though not variable declarations). This loosening requires a much more complicated enforestation algorithm that can track a prefix through partially enforested operators. It is unclear that this additional expressiveness afforded to infix macros is worth the added complexity to enforestation and future versions of the implementation will more closely align with the infix design presented in the formalism here.

Also worth noting, since the prefix is a sequence of expanded terms, any macros that appear before an infix macro will have already been expanded by the time the infix macro is invoked. This behavior introduces an asymmetry between the kinds of syntax an infix macro can match.

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

# Chapter 5

# Hygiene

Maintaining hygiene during macro expansion is perhaps the single most critical feature of an expressive macro system. Hygiene roughly means that an invocation of a macro does not introduce or use bindings that clash with what the programmer expects in the context of the invocation. Hygiene was worked out primarily in the Scheme community over many years [Kohlbecker et al., 1986, Bawden and Rees, 1988, Clinger, 1991, Dybvig et al., 1992]. The description of hygiene given here is purposefully vague because to date there has not been a crisp formal theorem for the fully general macros of Racket or sweet.js, though there have been formalizations for typed subsets of Scheme [Herman, 2010] and steps towards a full non-operational formalization [Adams, 2015].

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.

This chapter first presents an overview from a user's perspective of the kinds of naming clashes that the hygiene condition guards against and then extends the expansion formalization from Chapter 4 with hygiene.

## 5.1 Hygienic Binding

At a high-level there are two kinds of potential name clashes that hygiene guards against: name clashes from bindings introduced by a macro invocation (discussed in this section) and name clashes from bindings referenced from a macro definition

**Figure 5.1: Normal Unhygienic Swap Expansion**

```
var foo = 10, bar = 20;
swap(foo, bar)
```

expands to ⇒

```
var foo = 10, bar = 20;
var tmp = foo;
foo = bar;
bar = tmp;
```

(discussed in the next section).

The classic example of hygiene protecting a binding introduced by a macro is the `swap` macro, which introduces a new binding `tmp` to perform the standard variable swapping operation:

```
macro swap {
    rule { ($a, $b) } => {
        var tmp = $a;
        $a = $b;
        $b = tmp;
    }
}
```

A normal use of `swap` will expand as shown in Figure 5.1. However, if the macro invocation context uses a variable with the same name as the temporary variable introduced by the macro there is a potential problem as seen in Figure 5.2.

In JavaScript, redeclarations of a variable are ignored (so this code only defines a single `tmp` variable) and thus this code does not actually perform the expected swapping operation.

In a hygienic system like sweet.js, the expansion algorithm gives the binding introduced by the macro invocation a unique name and no clash occurs:

```
var tmp = 10, bar = 20;
var tmp2 = tmp;
```

**Figure 5.2: Bad Unhygienic Swap Expansion**

```
var tmp = 10, bar = 20;
swap(tmp, bar);
```

expands to ⇒

```
var tmp = 10, bar = 20;
var tmp = tmp;
tmp = bar;
bar = tmp;
```

```
tmp = bar;
bar = tmp2;
```

If clashes of this type were the only kind of potential binding problems the hygiene algorithm could be simple; hygiene could give all variable bindings introduced by a macro invocation a globally unique name. However, hygiene also involves references in macro definitions, which complicates the algorithm.

## 5.2 Hygienic References

The definition of a macro may contain references to variable bindings declared in the scope of the macro definition and those references must be consistent regardless of the invocation context.

For example, consider a simple `logger` macro that uses the builtin JavaScript function `console.log` to log a message:

```
macro logger {
    rule { ($msg ...) } => {
        console.log($msg ...)
    }
}
```

A macro author expects `console` to always be lexically bound from the macro definition site. Absent of the hygiene algorithm however, the variable `console` could be

**Figure 5.3: Unhygienic Console Expansion**

```
function sympathy(console) {
    logger('attempting to help out...');
    console('you are amazing!');
}
```

expands to ⇒

```
function sympathy(console) {
    console.log('attempting to help out...'); // wrong console!
    console('you are amazing!');
}
```

rebound. Consider a `sympathy` function that accepts a `console` function used to send encouragement. The unhygienic expansion of `logger` in the `sympathy` function is shown in Figure 5.3. In a non-hygienic system, the `console` bound by the function captures the `console` reference introduced by the `logger` macro.

In sweet.js and other hygienic systems the references are kept distinct via renaming:

```
function sympathy(console2) {
    console.log('attempting to help out...');
    console2('you are amazing!');
}
```

Unhygienic macros are not true abstractions since they force both macro authors and macro users to reason about all of the potential names that could be in all scopes. Hygiene allows users to reason about macros in the same way they reason about standard lexical scoping rules.

## 5.3 Macro Scoping

Macros in the Lisp and Scheme tradition are bound and scoped in an analogous fashion to other binders in the language. By following the scoping rules of the language,

**Figure 5.4: PI Expansion**

```
#define PI 3.14

main()
{
    char PI[] = "cherry";
}
```

expands to ⇒

```
#define PI 3.14

main()
{
    char 3.14[] = "cherry";
}
```

macros can shadow and be shadowed by other definitions in a program. This is in contrast to macro systems that work solely on lexical substitution like the C preprocessor (cpp) that do not respect scoping rules for macro definitions. For example, Figure 5.4 attempts to declare a variable `PI` where a macro with the same name has already been defined. This obviously results in a parse error. This expansion result happens because cpp simply performs lexical substitution; cpp does not take the scoping rules of C into account when performing its substitution. A more powerful macro system that integrates into the binding and scoping rules of the language would allow this example to do the more intuitive thing of rebinding `PI` in the function scope.

In Racket the above example can be written:

```
(define-syntax-rule (PI)
  3.14)

(define (main)
  (let ([PI "cherry"])
    PI))
```

Because the macro system is scoping aware, the `let` binding of `PI` inside the `main` procedure shadows the macro declared in the surrounding scope and a call to the procedure results in `"cherry"` as expected.

### 5.3.1 Racket Binding Forms

Racket has two basic binding forms for run-time values (with a fair number of variations): the `define` form and the `let` form. The `let` form is straightforward and scopes its binders to its body:

```
(let ([PI "cherry"])
  (string-append PI " pie"))
; evaluates to "cherry pie"
```

The `define` form depends on the context in which it is used (the top-level context, internal definition context, etc.) but it roughly means bind in the surrounding scope. So, in an internal definition context (an internal definition context is a context in which expressions and definitions can mix such as the body of a function) like the following, the `PI` binding is scoped to the function `make-pi`:

```
(define (make-pi)
  (define PI "cherry")
  (string-append PI " pie"))
(make-pi)
; evaluates to "cherry pie"
```

Racket also provides the corresponding binders `let-syntax` and `define-syntax` that bind macros according equivalent scoping rules as the binding forms for run-time values.

### 5.3.2 Sweet.js Binding Forms

The binding form situation in JavaScript is not quite as conceptually clean as in Racket. The main binding forms in JavaScript are the function declaration form and the `var` declaration form. Function declarations and `var` are similar to Racket's `define` in that the value is bound in the surrounding scope and declarations and other forms can be freely mixed.

However, they diverge in some important ways. Variable declarations are scoped to the surrounding function scope and are hoisted out of surrounding blocks:

```
function foo() {
  if (true) {
    var x = 42
  }
  return x;
}
foo() // evaluates to 42
```

In addition, redeclaring the same name is not an error in JavaScript and does not create a new binding; a second variable declaration of the same name is treated as a no-op:

```
function foo() {
  var x = 42;
  var x = "bar";
  return x;
}
foo() // evaluates to "bar"
```

In sweet.js there are two macro binding forms: a macro declaration and a let-bound macro declaration. Conceptually the macro declaration form behaves equivalently to a function declaration form where the macro is bound in the surrounding scope and in its template.

```
var id = "bar";
function foo() {
    // the id macro is bound inside the function foo
    macro id {
        rule { $x } => { $x }
    }
    return id 42
}
id // bound to the string "bar"
```

A let-bound macro behaves similarly except that the macro is not bound in its template. Let-bound macros allow macro authors to override builtin syntax forms like `function`:

**Figure 5.5: Mixed Definitions Expansion**

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

expands to ⇒

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

```
let function = macro {
    rule { $name ($params ...) { $body ...} } => {
        function $name($params ...) {
            console.log("calling the function");
            $body ...
        }
    }
}
```

A normal macro declaration form here would have resulted in an infinite expansion loop.

Interestingly, the particular syntax of JavaScript places limitations on macro binding forms that are not present in Racket. The expansion algorithm in Racket allows internal definitions and macro uses to be freely mixed in a given scope. This means that a macro can be used in an internal definition before the macro's definition as seen in Figure 5.5

In order to support mixing use and definition, the Racket expander must make multiple passes through a scope. The first pass discovers and loads macro definitions but crucially defers expansion in the right-hand side of an internal definition since there may be uses of macros that have not yet been loaded. The second pass expands inside the right-hand side of internal definition using the macros discovered during the first pass.

For example, after the first pass of expansion, the example in Figure 5.5 will become:

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

Here, the `id` macro has been loaded into the compile-time environment. During the second pass, the expander will expand inside of internal definitions, so the example becomes:

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

Critical for Racket's ability to mix internal definitions and macro uses is the fully delimited nature of internal definitions. The expander can skip over all of the syntax inside of the internal definition because there actually is an *inside* to skip over. However, this is not true for JavaScript since `var` statements are not delimited. It is thus not possible to use a macro that has not yet been defined in a `var` statement in sweet.js.

For example, this will fail:

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

When the expander reaches the `var` statement during the first pass, it has not yet loaded the `id` macro into the compile-time environment, so `id 100` will be left unexpanded. Since `id 100` is not a valid expression, this will fail to parse.

Note that at first glance it might appear that the line-ending 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 consumed by a macro. Second, the JavaScript specification calls for missing semicolons to be automatically inserted by the parser, so it is not guaranteed that a semicolon will close every `var` statement.

Though the expander cannot defer expansion of `var` statements, it still does

**Figure 5.6: Macro After Function Expansion**

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

expands to ⇒

```
function foo() {
  return 100;
}
```

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 as seen in Figure 5.6

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 is when writing mutually recursive macros, which is fully supported with the sweet.js approach.

## 5.4 Hygiene Implementation for sweet.js

The standard hygiene algorithm from Scheme and Racket is the mark and rename technique [Dybvig et al., 1992]. The key idea of mark and rename is that every identifier carries with it a list of potential renamings. Every binding form (e.g., a function declaration) pushes a new renaming to each identifier in its scope.

So in the following example, the outer `foo` function renames its parameter to `x1` and pushes that renaming down to everything in its body.

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

The `bar` function does the same with its parameter, renaming it to `x2`, which takes precedence over the `x1` renaming and we get the final expanded code:

```
function foo(x1) {
    function bar(x2) {
        return x2;
    }
    return x2;
}
```

Syntax objects must carry all of the renaming contexts until expansion has fully completed because it is not possible to know which renaming to use until all contexts have been applied.

So far the process described is basically standard lexical scoping but macro invocations complicate this by combining identifiers from the macro use-site and the macro definition site. To distinguish these identifiers, the expander also uses *marks*, which are added to syntax that is introduced by a macro invocation. Resolving the binding of an identifier then takes into account both the marks and renames on the syntax object when deciding which renaming to use.

The mark and rename approach as described so far is already rather complicated but it gets even more complicated when dealing with Racket's internal definitions (similar to variable declarations in JavaScript). To handle internal definitions Racket uses *definition contexts* [Section 3.8] [Flatt et al., 2012], which are difficult to reason about and hard to implement correctly or efficiently.

Motivated by the complexity of the hygiene algorithm, Matthew Flatt recently invented an alternate technique for tracking hygienic bindings in Racket based on sets of scopes [Flatt, 2015] that simplifies the hygiene algorithm.

While sweet.js initially used the mark, rename, and definition context approach, it now uses an adapted set-of-scopes algorithm, which I describe in the following sections.

## 5.5 Scopesets Overview

The key idea of scopesets is that during expansion each syntax object carries a set of the scopes in which it appears. Syntax forms that create a new scope (like function declarations) just add a fresh scope to each syntax object in its body. For example in the following example, $a$ is the top-level scope and $b$ is the scope created for the function `foo`.

```
var x^{a} = 42;
var y^{a} = 24;
function foo^{a}(x^{a,b}) {
    return x^{a,b} + y^{a,b};
}
```

In addition, there is a global *binding map* that associates each variable with a set of scopeset and binding pairs, where a binding is a fresh variable. The above snippet for example would have the following binding map where $x_1$, $x_2$, $y_1$, and $foo_1$ are all fresh variables.

$$
\begin{aligned}
&x \to (\{a\}, x_1), (\{a, b\}, x_2)\\
&y \to (\{a\}, y_1)\\
&\mathit{foo} \to (\{a\}, \mathit{foo}_1)
\end{aligned}
$$

When expansion finishes, the code generator will use the binding map to replace variable with its appropriate binding from the binding map.

```
var x_1 = 42;
var y_1 = 24;
function foo_1(x_2) {
      return x_2 + y_1;
}
```

Note that the `y` inside of the function body has a different scopeset ($\{a, b\}$) than the `y` at the top-level ($\{a\}$). When retrieving a binding from the binding map for a given identifier, the algorithm picks the scopeset/binding pair where the scopeset is the biggest subset of the identifier's scopeset. So, when retrieving the binding for `y`$^{\{a,b\}}$ inside the function body, $\{a\}$ is the biggest subset of $\{a, b\}$ and so it resolves to the top-level binding for `y`.

Macros of course are what make hygiene interesting. The following is a simple example with a macro definition and invocation.

```
var x^{a} = 42;
macro m^{a} {
    rule { $y } => {
        function foo^{a}(x^{a}) {
            return x^{a} + $y;
        }
    }
}
m x^{a};
```

If we expand this example as described so far, we run into a problem.

```
var x^{a} = 42;
function foo^{a}(x^{a}) {
    return x^{a} + x^{a};
}
```

Here, $a$ is the top-level scope and $b$ is the scope for the function body. What we wanted to happen is for the `x` binding introduced by the macro to be distinct from the `x` provided to the macro. However, the `x` provided to the macro was captured by the `x` binding introduced by the macro.

Readers familiar with traditional hygiene algorithms can probably guess that the way scopsets addresses this problem is by introducing a new scope for macro invocation; syntax freshly generated by the macro is marked with a new macro invocation scope. Similar to earlier mark and rename approaches, we mark the syntax provided to the macro invocation along with the syntax returned by the macro invocation. Double scopes are made to cancel out so only the syntax created by the macro carries the invocation scope. So, immediately after the macro is invoked we get:

```
var x^{a} = 42;
function foo(x^{a,b}) {
    return x^{a,b} + x^{a};
}
```

Here $b$ is the macro invocation scope. Then, expansion applies the function scope $c$ to the body of the function and we get:

```
var x$^{\{a\}}$ = 42;
function foo(x$^{\{a,b,c\}}$) {
    return x$^{\{a,b,c\}}$ + x$^{\{a,c\}}$;
}
```

Since $\{a, b, c\}$ is not a subset of $\{a, c\}$ but $\{a\}$ is, we get the correct bindings that distinguish the two `x` identifiers.

Unlike the previous mark and rename approach, we also must keep track of syntax provided to a macro invocation along with the syntax returned from a macro invocation. To see why, consider the following:

```
macro m {
    rule { $x } => {
        function foo(x$^{\{a\}}$) {
            function bar($x) {
                return x$^{\{a\}}$;
            }
        }
    }
}
m x$^{\{a\}}$
```

Here, the `m` macro expands to two nested functions `foo` and `bar` where the parameter identifier for `bar` is taken from the macro use-site. If we apply $b$ as the scope for the macro invocation, $c$ as the scope for the `foo` function, and $d$ as the scope for the `bar` function we run into an ambiguity:

```
function foo(x$^{\{a,b,c\}}$) {
    function bar(x$^{\{a,c,d\}}$) {
        return x$^{\{a,b,c,d\}}$;
    }
}
```

Note that the scopesets for both $\texttt{x}^{\{a,b,c\}}$ and $\texttt{x}^{\{a,c,d\}}$ are subsets of the same length for $\texttt{x}^{\{a,b,c,d\}}$. The intent of course is to bind the inner reference of `x` to the binding occurrence at `foo`.

The solution is to mark the syntax provided to the macro with use-site scopes. Now the invocation of `m` will look like $\texttt{m x}^{\{a,e\}}$ where $e$ is the use-site scope. The full expansion of our running example is the following.

```
function foo(x$^{\{a,b,c\}}$) {
    function bar(x$^{\{a,c,d,e\}}$) {
        return x$^{\{a,b,c,d\}}$;
    }
}
```

Note that x$^{\{a,c,d,e\}}$ is not a subset of x$^{\{a,b,c,d\}}$ and thus the ambiguity is resolved.

However, the introduction of use-site scopes causes a problem for macros that expand into declarations. For example,

```
macro def {
    rule { $x } => { var $x; }
}
def x;
x = 42;
```

The intent with this example is that since `x` is provided to the `def` macro, `x` should be bound at `x = 42`. However, the addition of the use-site scope makes the two `x` identifiers distinct:

```
var x$^{\{a,b\}}$;
x$^{\{a\}}$ = 42;
```

The solution to this problem is to keep track of the use-site scopes introduced during expansion and remove them from identifiers placed in a declaration position.

## 5.6 Ambiguous Scopeset Errors

Note that it is possible to write macros that result in ambiguous scopeset errors. Ambiguous scopeset errors occur when multiple "biggest subsets" are returned from the binding map in `resolve`. To illustrate these errors, consider the following example adapted from [Flatt, 2015] that uses var declarations and a macro-defining-macro.

```
macro varViaMacro {
    rule { $m $givenM } => {
        var x = 1;
        macro $m {
            rule { } => {
                // var name comes from the original use-site
                var $givenM = 2;
                // which declaration should x be bound to?
                x;
            }
        }
    }
}
varViaMacro m x;
m
```

Here the `varViaMacro` macro expands to a variable declaration for `x` and another macro declaration whose name is supplied by the invocation of `varViaMacro` that expands to a user-supplied var declaration along with a reference to `x`. After fully expanding this example we have the following scopesets:

var $\mathtt{x}^{\{a,b\}}$ = 1;
var $\mathtt{x}^{\{a,c\}}$ = 2;
$\mathtt{x}^{\{a,b,c\}}$;

The scopes in this example are `a` for the top-level scope, `b` for the expansion scope of the `varViaMacro` macro, and `c` for the expansion scope of the `m` macro. The problem is that the scopesets for both declarations of `x` are equal size subsets of the `x` reference and thus it is ambiguous which binding should be chosen.

As [Flatt, 2015] points out, this macro is, in some sense, ambiguous as to which binding should be chosen so an ambiguous error is correct. The prior hygiene algorithm with definition contexts did not throw these kinds of ambiguous binding errors. Rather, in the example given here, the `var x = 1` declaration would always be chosen. Flatt reports that while constructed examples like this one demonstrate a difference between the scopeset and definition context based hygiene algorithms, no practical examples of rule macros that give rise to ambiguous scopeset errors have been found.

## 5.7 Formal Hygienic Semantics

To solidify the overview of scopesets just presented, this section extends the semantics of Chapter 4 with scopesets.

To support hygiene, we first extend the grammar in Figure 5.7 and add a scope-

**Figure 5.7: Grammar for Hygienic Expansion**

| | | | |
|---|---|---|---|
| $x,\ name$ | $\in$ | $Variable$ | **Variable** |
| $t \in Token$ | $::=$ | $x \mid n \mid bop \mid \underline{(S)} \mid \underline{\{S\}}$ | **Token Tree** |
| $s \in Syntax$ | $::=$ | $t^C$ | **Syntax Object** |
| $id \in Identifier$ | $::=$ | $x^C$ | **Identifier** |
| | | | |
| $c$ | $\in$ | $Scope$ | **Scope** |
| $C \in ScopeSet$ | $=$ | $2^{Scope}$ | **Lexical Context** |
| | | | |
| $m$ | $::=$ | $ExprTerm \mid StmtTerm$ | **Term** |
| $ExprTerm$ | $::=$ | $id$ | Identifier |
| | | $n$ | Numeric Literal |
| | | $m\ op\ m$ | Binary Operator |
| | | $m\ \underline{(S)}$ | Call |
| | | `function` $id\ \underline{(S)}\ \underline{\{S\}}$ | Function Declaration |
| $StmtTerm$ | $::=$ | `var` $id$ `=` $m$ | Variable Declaration |
| | | `if` $\underline{(S)}\ \underline{\{S\}}$ | If Statement |
| | | `macro` $id\ \underline{\{S\}}$ | Macro Declaration |
| | | | |
| $S$ | $::=$ | $s^*$ | **Syntax Sequence** |
| $M$ | $::=$ | $m^*$ | **Term Sequence** |
| $a$ | $::=$ | $left \mid right$ | **Associativity** |
| $binary$ | $::=$ | $+ \mid - \mid * \mid / \mid \ldots$ | **Binary Operator Symbols** |
| $bop$ | $::=$ | $binary^{a,n}$ | **Binary Operators** |

**Figure 5.8: Hygienic Expansion Environment**

$$
\begin{array}{rcl}
E \in \mathit{Env} & ::= & \mathit{Name} \rightarrow_p \mathit{Transform} \\
B \in \mathit{BindingMap} & ::= & \mathit{Variable} \rightarrow_p 2^{(\mathit{ScopeSet},\, \mathit{Name})} \\
\\
\mathit{Transform} & ::= & \mathbf{Var}(\mathit{id}) \mid \mathit{PrimTrans} \mid \mathit{MacroTrans} \\
\mathit{PrimTrans} & ::= & \mathbf{VarDecl} \mid \mathbf{FunDecl} \mid \mathbf{MacroDecl} \mid \mathbf{IfStmt} \\
\\
\mathit{MacroTrans} & ::= & (\mathit{Match}, \mathit{Trans}) \\
\mathit{Match} & ::= & (\mathit{Term}^*, \mathit{Syntax}^*) \rightarrow (\mathit{Subst}, \mathit{Term}^*, \mathit{Syntax}^*) \\
\mathit{Trans} & ::= & \mathit{Subst} \rightarrow \mathit{Syntax}^* \\
\\
\mathit{Subst} & ::= & \mathit{Variable} \rightarrow_p \mathit{Syntax}^*
\end{array}
$$

set to syntax objects. A scopeset ($C$) is a set of scopes ($\{c_1, \dots, c_n\}$) where a scope is an element drawn from some recursively enumerable set (for example, an implementation might simply use numbers to represent scopes).

Scopes by themselves do not represent any binding information. Rather, binding information is stored in the *BindingMap* map that associates variables with a set of scopeset/name pairs. The name part of the pair represents the binding while the scopeset records the scopes in which the binding occurs.

To get a binding for a syntax object, we now use the `resolve` function from Figure 5.9 that looks up a syntax object's binding in the binding map. The key idea of `resolve` is that it looks up the scopeset/name pairs for the given syntax object and then returns the name from the pair who's scopeset is the biggest subset of the syntax object's scopeset.

For example, if the binding map for $x$ contains the pairs $(\{a\}, \mathit{name}_1)$ and $(\{a, b\}, \mathit{name}_2)$, `resolve` of $x^{\{a,b,c\}}$ would return $\mathit{name}_2$ since $\{a, b\}$ is the biggest subset of $\{a, b, c\}$ while `resolve` of $x^{\{a,c\}}$ would give $\mathit{name}_1$ since $\{a\}$ is the biggest subset of

**Figure 5.9: Resolve Function**

$$
\begin{array}{lcl}
\texttt{resolve} : (\mathit{BindingMap}, \mathit{Identifier}) \to \mathit{Name} & & \\
\texttt{resolve}(B, x^C) & = & x \\
\quad \text{if } x \notin \mathit{dom}(B) & & \\
\texttt{resolve}(B, x^C) & = & \mathit{name}_i \\
\quad \text{if } \{(C_1, \mathit{name}_1), \ldots, (C_n, \mathit{name}_n)\} = B(x) & & \\
\qquad C_i = \texttt{biggestSubset}(C, \{C_1, \ldots, C_n\}) & & \\
\qquad (C_i, \mathit{name}_i) \in B(x) & &
\end{array}
$$

$\{a, c\}$.

### 5.7.1 Hygienic Enforest

The enforest relation in Figure 5.10 is now parameterized by the binding map along with the compile-time environment and current precedence level.

As one might expect, the biggest change is to the rule that handles macro invocation. After pulling the $(\mathit{match}, \mathit{trans})$ pair out of the compile-time environment, enforest creates two new scopes $c$ and $c_{use}$ where $c$ is used to distinguish syntax generated by the macro and $c_{use}$ is used to distinguish syntax provided to the macro (i.e., $\underline{\text{use}}$d by the macro). Both scopes are applied via the `mark` function defined in Figure 5.11 to the substitution environment $\theta$ but then only the $c$ scope is applied to the result of *trans*. Like marks in the old hygiene algorithm two scopes cancel out so by applying $c$ twice, syntax that has a $c$ scope must have been generated by the macro rather than be provided by it. Since the $c_{use}$ scope is applied only to the substitution environment, syntax with a $c_{use}$ scope must have been present at the macro use-site.

Since we need to remove use-site scopes from identifiers that wind up in definitions, we also add the $c_{use}$ scope to the set of use-site scopes ($C$). The use-site scope set will be used in `expandTokens` described in the next section.

**Figure 5.10: Hygienic Enforest Function**

$$
\begin{array}{rcl}
M,\ \bot,\ id \cdot S,\ C & \rightarrow_{n,E,B} & M,\ id,\ S,\ C \\
M,\ \bot,\ n_1 \cdot S,\ C & \rightarrow_{n,E,B} & M,\ n_1,\ S,\ C \\
M,\ id_1,\ S,\ C & \rightarrow_{n,E,B} & M,\ id_2,\ S,\ C \\
 & & \text{if } \mathbf{Var}(id_2) = E(\texttt{resolve}(id_1,\ B)) \\
M,\ id,\ S,\ C & \rightarrow_{n,E,B} & M',\ \bot,\ \texttt{mark}(S_{res},\ c) \cdot S', C \cup \{c_{use}\} \\
 & & \text{if } (\mathit{match}, \mathit{trans}) = E(\texttt{resolve}(id,\ B)) \\
 & & c, c_{use}\ \mathit{fresh} \\
 & & (\theta, M', S') = \mathit{match}(M, S) \\
 & & S_{res} = \mathit{trans}(\texttt{mark}(\theta,\ \{c, c_{use}\})) \\
M,\ id_{var},\ id \cdot \texttt{=} \cdot S,\ C & \rightarrow_{n,E,B} & M,\ \texttt{var}\ id\ m,\ S', C' \\
 & & \text{if } \mathbf{VarDecl} = E(\texttt{resolve}(id_{var},\ B)) \\
 & & \epsilon,\ \bot,\ S,\ C \rightarrow^{*}_{0,E,B} \epsilon,\ m,\ S', C' \\
 & & m \in \mathit{ExprTerm} \\
M,\ id_{fun},\ id_1 \cdot \underline{\texttt{(}id_2\texttt{)}} \cdot \underline{\texttt{\{}S_1\texttt{\}}} \cdot S_2,\ C & \rightarrow_{n,E,B} & M,\ \texttt{function}\ id_1\ \underline{\texttt{(}id_2\texttt{)}}\ \underline{\texttt{\{}S_1\texttt{\}}},\ S_2,\ C \\
 & & \text{if } \mathbf{FunDecl} = E(\texttt{resolve}(id_{fun},\ B)) \\
M,\ id_{if},\ \underline{\texttt{(}S_1\texttt{)}} \cdot \underline{\texttt{\{}S_2\texttt{\}}} \cdot S_3,\ C & \rightarrow_{n,E,B} & M,\ \texttt{if}\ \underline{\texttt{(}S_1\texttt{)}}\ \underline{\texttt{\{}S_2\texttt{\}}},\ S_3,\ C \\
 & & \text{if } \mathbf{IfStmt} = E(\texttt{resolve}(id_{if},\ B)) \\
M,\ id_{mac},\ id \cdot \underline{\texttt{\{}S_1\texttt{\}}} \cdot S_2,\ C & \rightarrow_{n,E,B} & M,\ \texttt{macro}\ id\ \underline{\texttt{\{}S_1\texttt{\}}},\ S_2,\ C \\
 & & \text{if } \mathbf{MacroDecl} = E(\texttt{resolve}(id_{mac},\ B)) \\
M,\ m,\ \underline{\texttt{(}S\texttt{)}} \cdot S',\ C & \rightarrow_{n,E,B} & M,\ m\ \underline{\texttt{(}S\texttt{)}},\ S',\ C \\
 & & \text{if } m \in \mathit{ExprTerm} \\
M,\ m_1,\ \mathit{bop}^{a,n_{op}} \cdot S,\ C & \rightarrow_{n,E,B} & M,\ m_1\ \mathit{bop}\ m_2,\ S', C' \\
 & & \text{if } n_{op} >_a n \\
 & & \epsilon,\ \bot,\ S,\ C \rightarrow^{*}_{n_{op},E,B} \epsilon,\ m_2,\ S', C' \\
 & & m_1 \in \mathit{ExprTerm} \\
 & & m_2 \in \mathit{ExprTerm}
\end{array}
$$

**Figure 5.11: Mark Function**

$$
\begin{array}{lcl}
\texttt{mark} : (\mathit{Syntax}^*, \mathit{Scope}) \rightarrow \mathit{Syntax}^* & & \\
\texttt{mark}(t^C \cdot S, c) & = & t^{C'} \cdot \texttt{mark}(S, c) \\
\quad \text{if } c \notin C & & \\
\qquad C' = C \cup \{c\} & & \\
\texttt{mark}(t^C \cdot S, c) & = & t^{C'} \cdot \texttt{mark}(S, c) \\
\quad \text{if } c \in C & & \\
\qquad C' = C \setminus \{c\} & & \\
\\
\texttt{mark} : (\mathit{Subst}, \mathit{Scope}) \rightarrow \mathit{Subst} & & \\
\texttt{mark}(\{(x_1, S_1), \dots (x_n, S_n)\}, c) & = & \{(x_1, \texttt{mark}(S_1, c)), \dots (x_n, \texttt{mark}(S_n, c))\}
\end{array}
$$

Our collection of expand functions are now much more interesting than in Chapter 4 and the reason for separating the work into three functions hopefully makes a little more sense. Each expand function now takes as additional parameters the binding map ($B$), a set of the use-site scopes ($C$), and a scope ($c$) to distinguish let-bound macros as I will describe in section 5.8.

## 5.7.2 Hygienic `expandTokens`

For `expandTokens` defined in Figure 5.13, the cases for macro, variable, and function declarations must all do similar work for hygiene. In particular, each case takes the scope set $C$ of the declaration's identifier and removes any use-site scopes. It then extends the binding map with a pair of the modified scopeset and a fresh binding and extends the compile-time environment with a fresh binding mapping to the meaning of the declaration (in the case of a macro declaration the meaning is the macro transformer resulting from `loadMacro` and for the others it is a var transformer with the declaration identifier with the use-site scopes removed).

**Figure 5.12: Hygienic Expand**

$$\texttt{expand} : (\mathit{Syntax}^*, \mathit{Env}, \mathit{BindingMap}, \mathit{Scopeset}) \to \mathit{Term}^*$$

$$\texttt{expand}(S,\ E,\ B,\ C_{use},\ c) \quad = \quad \texttt{expandTerms}(M,\ E',\ B',\ C'_{use},\ c)$$
$$\text{if } \texttt{expandTokens}(\epsilon,\ S,\ E,\ B,\ C_{use},\ c) = (M,\ E',\ B', C'_{use})$$

The `expandTokens` function does this hygiene work for the flat declarations because the meaning for each declaration must be immediately loaded into the compile-time environment. In contrast, the nested bindings in a function parameters is deferred to the `expandTerms` function (expansion does not move into nested delimiters until `expandTerms`).

### 5.7.3 Hygienic `expandTerms`

In `expandTerms` defined in Figure 5.14, the interesting case that has changed is function declarations. The function case creates a new scope $c'$ and extends the function parameter's scopeset with the new scope along with every syntax object in the function's body. Then the binding environment and compile-time environment are extended with the new bindings.

## 5.8 Non-Recursive Macro Declaration

A recurring challenge in hygienic macro system design is flat binding forms. Nested binding forms, such as function declarations, separate out syntax where the bindings should apply (the function body) from the bindings themselves (the parameter list). Flat binding forms, such as variable declarations, mix bindings with the syntax where the bindings should apply. In earlier approaches [Flatt et al., 2012] definition contexts were used to handle flat binding forms. While scopesets have simplified some aspects of flat binding forms, implementing a flat non-recursive macro declaration form presents a challenge.

**Figure 5.13: Hygienic Expand Tokens**

$\texttt{expandTokens} : (\mathit{Term}^*, \mathit{Syntax}^*, \mathit{Env}, \mathit{BindingMap}, \mathit{Scopeset})$
$\quad\quad \to (\mathit{Term}^*, \mathit{Env}, \mathit{BindingMap}, \mathit{Scopeset})$

$\texttt{expandTokens}(M,\, \epsilon,\, E,\, B,\, C_{use},\, c) \quad = \quad (\texttt{reverse}(M), \Sigma, B, C_{use})$

$\texttt{expandTokens}(M,\, S,\, E,\, B,\, C_{use},\, c) \quad = \quad \texttt{expandTokens}(m_{res} \cdot M',\, S',\, E',\, B',\, C'_{use},\, c)$
  if $M,\, \bot,\, S, C_{use} \to^*_{0,E,B} M',\, \texttt{macro}\ x^C\ \underline{\texttt{\{}S_1\texttt{\}}},\, S', C'_{use}$
    $C' = C \setminus C'_{use}$
    $\mathit{name}$ is fresh
    $B' = B[x := (C', \mathit{name})]$
    $E' = E[\mathit{name} := \texttt{loadMacro}(S_1)]$
    $m_{res} = \texttt{macro}\ x^{C'}\ \underline{\texttt{\{}S_1\texttt{\}}}$

$\texttt{expandTokens}(M,\, S,\, E,\, B,\, C_{use},\, c) \quad = \quad \texttt{expandTokens}(m_{res} \cdot M',\, S',\, E',\, B',\, C'_{use},\, c)$
  if $M,\, \bot,\, S, C_{use} \to^*_{0,E,B} M',\, \texttt{var}\ x^C\ \texttt{=}\ m,\, S', C'_{use}$
    $C' = C \setminus C'_{use}$
    $\mathit{name}$ is fresh
    $B' = B[x := (C', \mathit{name})]$
    $E' = E[\mathit{name} := \texttt{Var}(x^{C'})]$
    $m_{res} = \texttt{var}\ x^{C'}\ \texttt{=}\ m$

$\texttt{expandTokens}(M,\, S,\, E,\, B,\, C_{use},\, c) \quad = \quad \texttt{expandTokens}(m_{res} \cdot M',\, S',\, E',\, B',\, C'_{use},\, c)$
  if $M,\, \bot,\, S, C_{use} \to^*_{0,E,B} M',\, \texttt{function}\ x^C\ \underline{\texttt{(}S_1\texttt{)}}\ \underline{\texttt{\{}S_2\texttt{\}}},\, S', C'_{use}$
    $C' = C \setminus C'_{use}$
    $\mathit{name}$ is fresh
    $B' = B[x := (C', \mathit{name})]$
    $E' = E[\mathit{name} := \texttt{Var}(x^{C'})]$
    $m_{res} = \texttt{function}\ x^{C'}\ \underline{\texttt{(}S_1\texttt{)}}\ \underline{\texttt{\{}S_2\texttt{\}}}$

$\texttt{expandTokens}(M,\, S,\, E,\, B,\, C_{use},\, c) \quad = \quad \texttt{expandTokens}(m_{res} \cdot M',\, S',\, E,\, B,\, C'_{use},\, c)$
  if $M,\, \bot,\, S, C_{use} \to^*_{0,E,B} M',\, m_{res},\, S', C'_{use}$

**Figure 5.14: Hygienic Expand Terms**

$$\texttt{expandTerms} : (\mathit{Term}^*, \mathit{Env}, \mathit{BindingMap}, \mathit{ScopeSet}, \mathit{Scope}) \to \mathit{Term}^*$$

$$\begin{aligned}
\texttt{expandTerms}(id \cdot M,\, E,\, B,\, C_{use},\, c) &= id \cdot \texttt{expandTerms}(M,\, E,\, B,\, C_{use},\, c)\\
\texttt{expandTerms}(n \cdot M,\, E,\, B,\, C_{use},\, c) &= n \cdot \texttt{expandTerms}(M,\, E,\, B,\, C_{use},\, c)\\
\texttt{expandTerms}(m \cdot M,\, E,\, B,\, C_{use},\, c) &= m \cdot \texttt{expandTerms}(M,\, E,\, B,\, C_{use},\, c)\\
&\text{if } m = m_1 \; op \; m_2\\
\texttt{expandTerms}(m \cdot M,\, E,\, B,\, C_{use},\, c) &= m \cdot \texttt{expandTerms}(M,\, E,\, B,\, C_{use},\, c)\\
&\text{if } m = \texttt{var}\; x \;\texttt{=}\; m_{init}\\
\texttt{expandTerms}(m \cdot M,\, E,\, B,\, C_{use},\, c) &= m' \cdot \texttt{expandTerms}(M,\, E,\, B,\, C_{use},\, c)\\
&\text{if } m = m_{fn}\; \underline{\texttt{(}S\texttt{)}}\\
&\quad M_1 = \texttt{expand}(S,\; E,\; B,\; C_{use},\, c)\\
&\quad m' = m_{fn}\; \underline{\texttt{(}M_1\texttt{)}}\\
\texttt{expandTerms}(m \cdot M,\, E,\, B,\, C_{use},\, c) &= m' \cdot \texttt{expandTerms}(M,\, E,\, B,\, C_{use},\, c)\\
&\text{if } m = \texttt{function}\; id_1\; \underline{\texttt{(}x^{C}_{arg}\texttt{)}}\; \underline{\texttt{\{}S_{body}\texttt{\}}}\\
&\quad c',\, name_{arg} \text{ are fresh}\\
&\quad C' = C \cup c'\\
&\quad B' = B[x_{arg} := (C', name_{arg})]\\
&\quad E' = E[name_{arg} := \mathbf{Var}(x^{C'}_{arg})]\\
&\quad S'_{body} = \texttt{mark}(S_{body},\, c')\\
&\quad M_{body} = \texttt{expand}(S'_{body},\; E',\; B',\; C_{use},\, c')\\
&\quad m' = \texttt{function}\; id_1\; \underline{\texttt{(}x^{C'}_{arg}\texttt{)}}\; \underline{\texttt{\{}M_{body}\texttt{\}}}\\
\texttt{expandTerms}(m \cdot M,\, E,\, B,\, C_{use},\, c) &= m' \cdot \texttt{expandTerms}(M,\, E,\, B,\, C_{use},\, c)\\
&\text{if } m = \texttt{if}\; \underline{\texttt{(}S_1\texttt{)}}\; \underline{\texttt{\{}S_2\texttt{\}}}\\
&\quad M_1 = \texttt{expand}(S_1,\; E,\; B,\; C_{use},\, c)\\
&\quad M_2 = \texttt{expand}(S_2,\; E,\; B,\; C_{use},\, c)\\
&\quad m' = \texttt{if}\; \underline{\texttt{(}M_1\texttt{)}}\; \underline{\texttt{\{}M_2\texttt{\}}}
\end{aligned}$$

**Figure 5.15: Let Macros**

$$\texttt{expandTokens}(M,\, S,\, E,\, B,\, C_{use},\, c) \quad = \quad \texttt{expandTokens}(M_{res} \cdot M',\, S',\, E',\, B',\, C'_{use},\, c)$$

if $M,\, \bot,\, S, C_{use} \rightarrow^{*}_{0,E,B} M',\, \texttt{let}\ x^{C}\ \texttt{= macro}\ \underline{\texttt{\{}S_1\texttt{\}}},\, S', C'_{use}$

$c'$ *fresh*

$S'_1 = \texttt{mark}(S_1,\, c')$

$C_1 = C \cup c'$

$C_2 = C \setminus c$

$m_1 = \texttt{macro}\ x^{C_1}\ \underline{\texttt{\{rule}\ \underline{\texttt{\{\}}}\ \texttt{=>}\ \underline{\texttt{\{}x^{C_2}\texttt{\}}}\texttt{\}}}$

$m_2 = \texttt{macro}\ x^{C}\ \underline{\texttt{\{}S'_1\texttt{\}}}$

$M_{res} = m_1 \cdot m_2$

The macro declarations form as described so far is recursive: the macro name is bound in its body. While useful, we also need a form that is not recursive. In Racket the non-recursive definition form is called `let-syntax` however the body in which the macro definition is bound is nested:

```
(let-syntax [(m ...)]
    (m 42))
```

Maintaining hygiene for `let-syntax` is straightforward; a new scope is applied only in the body of `let-syntax` and not in the initializer, which prevents `m` from being bound inside of its definition.

We could use a similar nested body approach in JavaScript. Perhaps the syntax could look something like:

```
let m = macro { ... } in {
    m ...
}
```

However, this kind of nested declaration form is alien to the syntax of JavaScript; a flat declaration form is needed instead. The challenge with a flat non-recursive form is making sure the macro name is bound in the surrounding scope but not in the nested macro body scope.

The approach we take is to define a non-recursive macro in terms of two recursive macros. In particular, when the expander gets to a let macro declaration where the current scope is $b$:

```
let m$^{\{a,b\}}$ = macro { ...$^{\{a,b\}}$ };
m ...
```

The expander will replace the definition with the following two recursive macro declarations:

```
macro m$^{\{a,b,c\}}$ { rule {} => { m$^{\{a\}}$ } };
macro m$^{\{a,b\}}$ { ...$^{\{a,b,c\}}$ };
m ...
```

The expander creates a new scope $c$ for the body of the macro and installs a new macro in the same scope that simply expands to the macro name but with the surrounding scope $b$ removed. That way, any occurrence of `m`$^{\{a,b,c\}}$ in the macro body will expand to `m`$^{\{a\}}$.

This is formalized in Figure 5.15 as a new case for `expandTokens`, which adds a fresh scope $c'$ to the macro body and sets up a new macro that expands to the macro name with the current scope $c$ removed.

## 5.9 Infix Macros and Hygiene

Infix macros add one additional complication to the hygiene algorithm. In particular, infix macros have the ability to manipulate declarations that have already been registered in the binding environment, such as moving a declaration into a different scope or removing a declaration entirely. For example, the macro in Figure 5.16 moves the `x` declaration into a new function scope leaving the `x` reference an orphan with no associated binding.

This behavior does not appear to be a practical concern since standard macros can also expand to orphan references. The difference is that the declaration bindings are not registered in the binding map for standard macros (since the macro is invoked before expansion loads declarations into the binding map). So, it seems the only real

**Figure 5.16: wrapBefore Expansion**

```
macro wrapBefore {
    rule infix { $prefix ... | } => {
        function wrapper() {
            $prefix ...
        }
    }
}
var x = 42;
wrapBefore
x;
```

expands to ⇒

```
function wrapper() {
    var x = 42;
}
x;
```

difference between infix and standard macros in this regard is that a badly behaved infix macro could pollute the binding map with orphan bindings.

# Chapter 6

# Application - Contracts

Large software systems typically consist of many modules (e.g., packages, classes, functions) produced by different development teams. When a system fails, an initial difficulty is *fault localization*: identifying the module that failed to perform as expected. Undocumented module interfaces are problematic for various reasons, not least because they lead to disagreements about which module is considered "at fault" and should be fixed.

Software engineers embrace behavioral contracts because they address many of these problems. In particular, behavioral contracts provide a mechanism to explicitly document each module's assumptions and guarantees; to dynamically detect contract violations; and to identify faulty modules. Behavioral contracts are widely used in procedural, object-oriented, and functional languages, including Eiffel [Meyer, 1992], C [Rosenblum, 1995], C# [McFarlane, 2002], Haskell [Hinze et al., 2006], Java [Karaorman et al., 1999], Python [Tuglular et al., 2009], Scheme [Findler and Blume, 2006, Dimoulas et al., 2011, Matthew Flatt and PLT, 2010], and SmallTalk [Carrillo et al., 1996].

In a dynamically typed language like JavaScript, tracking down the cause of a bug can be particularly frustrating. Often simple type violations that could have been caught by a type system slip through. For example, if `o` is undefined then attempting to look up a property on it via `o.foo` will result in the runtime error:

```
TypeError: Cannot read property 'foo' of undefined
```

While the error is descriptive enough for the proximate cause of the failure (attempting to use an `undefined` value), it does not provide information about the ultimate cause (i.e., which piece of the code produced the `undefined` value in the first place). The actual party at fault might be in some function or file not even in the stack trace. You have to recreate the control flow in your head or jump into the debugger just to figure out who to blame.

Behavioral contracts provide a way to specify invariants that are checked at runtime and provide precise error reporting by blaming the correct module or section of code responsible for the invariant violation. The invariants that contracts can enforce are similar to the kinds of safety properties a type system can provide. Note that a failure-free execution only holds for that execution; a second execution with different inputs might trigger a contract violation.

This chapter describes contracts.js, a behavioral contract system for JavaScript that provides declarative specification via macros. Contracts.js is implemented as a runtime library that wraps JavaScript functions in contracts that enforce the specified behavior along with a set of macros that ease the burden of specifying the program behavior.

In the next section I present an overview of how contracts.js can be used along with its primary features and then discuss how the library and macros are constructed.

## 6.1 Contracts.js Overview

This section gives a user-centered overview of how contracts.js works. To start off, consider a simple object that represents cats.

```
var spot = {
    name: "Spot",
    age: 3,
    haz: "cheezburger"
};
```

Along with our cat data representation we can also have a simple function `isVegetarian` that checks the food habits of cats:

```
function isVegetarian(o) {
    // without loss of generality
    // only cheezburgers are meat
    return o.haz !== "cheezburger";
}
isVegetarian(spot);  // false
```

Now consider attempting to invoke `isVegetarian` with a faulty cat object.

```
isVegetarian({
    name: "Tiger",
    age: 2
});
```

Calling this function will give us `true` (since `o.haz` is `undefined`, which is not `"cheezburger"`) but this result is clearly not intended. Note that this behavior is potentially worse than a confusing error message since we get a result that is just subtly wrong.

Contracts help us to address this problem by stating and enforcing what kinds of data our functions work on. In this case the `isVegetarian` function must be called with an object with a `haz` string property and it must return a boolean. So, using contracts.js we can apply that contract like so:

```
@ ({haz: Str}) -> Bool
function isVegetarian(o) {
    return o.haz !== "cheezburger";
}
```

The syntax for putting a contract on a function is:

```
@ (...) -> ...
function name(...) {
    ...
}
```

Contracts for each argument to the function go in the parentheses to the left of the `->` and the contract for the return value of the function goes on the right.

So in our `isVegetarian` example the argument contract is `{haz: Str}` (meaning an object with a `haz` string property) and the return value contract is `Bool` for boolean values.

Note that the object contract is only checking for the `haz` property, not every property that a cat object might have (i.e., this is structural typing). Other “haz”-able objects will work too.

Now if we invoke `isVegetarian` with a faulty object, contracts.js will throw a detailed exception helping the programmer to pinpoint exactly where the fault is located.

```
Error: isVegetarian: contract violation
expected: Str
given: undefined
in: the haz property of
    the 1st argument of
    ({haz: Str}) -> Bool
function isVegetarian guarded at line: 2
blaming: (calling context for isVegetarian)
```

The contract violations reports both the expanded and actual value along with appropriate line number information. It is even blaming the correct part of the code (the caller to `isVegetarian` was at fault for supplying a bad cat object).

So to make this running example a little more interesting, consider a richer representation for the kinds of things a cat can "haz".

```
var spot = {
    name: "Spot",
    age: 3,
    haz: ["cheezburger", "dataz", "iphonez", "fwend"]
};
```

Now that the `haz` property is an array, we also need to update the `isVegetarian` function and associated contract appropriately.

```
@ ({haz: [...Str]}) -> Bool
function isVegetarian(o) {
    for (var i = 0; i < o.haz.length; i++) {
        if (o.haz[i] !== "cheezburger") {
            return false;
        }
    }
    return true;
}
```

Here, the contract `[...Str]` means that the value must be an array of strings. Contracts.js also allows the programmer to specify a different contract for each index of the array by not using the ellipses (e.g., `[Str, Num, Bool]` is the contract for an array like `["foo", 42, true]`).

Now if the revised `isVegetarian` is invoked with a faulty object, contracts.js will throw the appropriately descriptive error.

```
isVegetarian({
    name: "Tiger",
    age: 2,
    haz: ["cheezburger", false]
});
```

The error message reads:

```
Error: isVegetarian: contract violation
expected: Str
given: false
in: the 1st field of
    the haz property of
    the 1st argument of
    ({haz: [....Str]}) -> Bool
function isVegetarian guarded at line: 2
blaming: (calling context for isVegetarian)
```

Note that the error message actually informs the user of the exact index of the array that failed the contract.

To generalize from our example so far, consider allowing the caller of `isVegetarian` to decide how to define vegetarianism by providing a predicate function:

```
@ ({haz: [...Str]}, (Str) -> Bool) -> Bool
function isVegetarian(o, isVeg) {
    var ret = true;
    for (var i = 0; i < o.haz.length; i++) {
        if (!isVeg(o.haz)) {
            ret = false;
        }
    }
    return ret;
}
```

Note that now our contract has a function argument contract `(Str) -> Bool` for the `isVeg` predicate. This contract means that the caller to `isVegetarian` must supply a function that when called with a string will return a boolean.

Attentive readers might have noticed a bug in the rewrite. Rather than calling the `isVeg` predicate with an individual `haz` element, the entire `haz` array is passed to `isVeg`. Thankfully, the contract will help us catch this bug.

```
isVegetarian({
    name: "Tiger",
    age: 2,
    haz: ["cheezburger", "dataz"]
}, function(val) {
    return val !== "cheezburger";
});
```

Attempting to invoke this code will cause the following error to be thrown.

```
Error: isVegetarian: contract violation
expected: Str
given: cheezburger,dataz
in: the 1st argument of
    the 2nd argument of
    ({haz: [....Str]}, (Str) -> Bool) -> Bool
function isVegetarian guarded at line: 2
blaming: function isVegetarian
```

Notice that the error blames the function `isVegetarian` instead of the caller. Blaming `isVegetarian` is correct because `isVegetarian` is the one that went wrong by invoking `isVeg` with the wrong type of argument.

Since the contract system helped us track down exactly where the fault was introduced we can fix up the faulty code with ease.

```
@ ({haz: [...Str]}, (Str) -> Bool) -> Bool
function isVegetarian(o, isVeg) {
    var ret = true;
    for (var i = 0; i < o.haz.length; i++) {
        if (!isVeg(o.haz[i])) {
            ret = false;
        }
    }
    return ret;
}
```

Now consider the case where the caller is at fault.

```
isVegetarian({
    name: "Tiger",
    age: 2,
    haz: ["cheezburger", "dataz"]
}, function(val) {
    val !== "cheezburger";
    // forgot the return keyword!
});
```

Again, contracts.js provides a descriptive error message that helps us to localize the problem.

```
Error: isVegetarian: contract violation
expected: Bool
given: undefined
in: the return of
    the 2nd argument of
    ({haz: [....Str]}, (Str) -> Bool) -> Bool
function isVegetarian guarded at line: 2
blaming: (calling context for isVegetarian)
```

Note that here blame correctly falls on the caller to `isVegetarian` for supplying a bad `isVeg` function. This may seem like a small thing but the ability to correctly ascribe blame is important as higher-order functions start to flow through your application. Without blame tracking the code at fault might not show up in either the error message or the stack trace causing the programmer to start looking in the wrong place for the bug.

## 6.2 Contracts.js Implementation

There are two important pieces of the implementation of contracts.js: the runtime library and the macros. Macros provide an expressive specification language for programmers to describe the behavior they want the program to adhere to while the runtime library is responsible for wrapping functions and objects in contracts that check and enforce the specification.

The core function of the runtime library is `guard` that takes a contract along with a value and returns the value wrapped in an appropriate contract.

```
var numId = guard(fun(Num, Num),
                  function (x) { return x });

numId(42);
numid("42"); // throws error
```

In the above example, `fun` and `Num` are contract *combinators* corresponding to contracts for functions and values that are `typeof` number respectively. A contract is built up by applying the appropriate combinators. Here, the usage of the `fun` combinator creates a contract that represents a function that must accept a number parameter and return a number.

Trying to write a program using just this library quickly becomes unwieldy. For example, consider the function below that takes two parameters, a number and a function.

```
var f = guard(
        fun([Num,
             fun([Str, Num],
                 object({ foo: Str,
                          bar: Num}))],
            Num),
        function f(x, h) {
            // ...
        });
```

Hopefully, this example illustrates the limitations of using the syntax tools built into JavaScript (i.e., just functions and function application) to represent a contract specification. Macros will allow us to provide a more readable syntax for users of the contract library.

## 6.3 Macro Implementation

At a high-level, the macro for contracts.js converts the contract syntax into the appropriate application of contract combinators along with an invocation of the `guard` wrapper function.

The macro identifier for wrapping a function in a contract is `@` which was picked since the use of `@` in a similar fashion has precedent other languages such as Python decorators [Smith et al.].

By using custom patterns, we can make the macro definition nicely compact and declarative.

```
macro @ {
    rule {
        $c:contract
        function $name ($params ...) { $body ... }
     } => {
         var $name = guard($c,
                           function $name($params ...) {
                               $body ...
                           })
    }
}
```

The pattern `$c:contract` uses the separately defined `contract` macro to recursively match the contract syntax and bind the result to `$c`. There are three kinds of contracts that the `contract` macro must match against: object contracts, function contracts, and primitive contracts. For each rule that the `contract` macro matches, it expands to the appropriate invocations of the runtime contract combinators. Note that the `contract` macro definition is recursively defined; each domain in a function contract, for example, can be any of the contracts.

```
macro contract {
    rule {
        {
            $($prop:ident : $c:contract) (,) ...
        }
    } => {
        object({$($prop : $c) (,) ...})
    }
    rule {
        ($dom:contract (,) ...) ->
        $rng:contract
    } => {
        fun([$dom (,) ...], $rng)
    }
    rule { $prim:ident } => {
        $prim
    }
}
```

For example, the `contract` macro would expand `{ foo: (Str) -> Bool }` to the combinator `object({foo: fun([Str], Bool)})`.

## 6.4 Additional Contract.js Features

In addition to the contract features described so far, contracts.js also supports a variety of declarative contract features such as dependent contracts, parametric polymorphism, and asynchronous contracts.

### 6.4.1 Dependent Contracts

Contracts.js also supports contracts that depend on the value of arguments to a function. Dependent contracts can be written as:

```
@ (x: Pos) -> res: Num | res > (Math.sqrt(x) - 0.1) &&
                         res < (Math.sqrt(x) + 0.1)
function square_root(x) { return Math.sqrt(x); }
```

Each argument must be named via the syntax `<name>: <contract>`. The named arguments can be referenced in the dependency guard, which is an expression following the pipe character. The guard is a predicate so if the guard evaluates to **`true`** the dependent function contract will pass, otherwise it fails.

Similar to the syntax for ES2015 arrows, a guard with more than a single expression can be written by surrounding a sequence of statements in a block:

```
@ (x: Pos) -> res: Num | {
    var fromlib = Math.sqrt(x);
    return res <= x && fromlib === res;
}
function square_root(x) { return Math.sqrt(x); }
```

Note that guards in a dependent contract could potentially violate a contract on one of the arguments:

```
@ (f: (Num) -> Num) -> res: Num | f("foo") > 10
function foo(f) { return f(24) }
```

Contracts.js follows *indy semantics* [Dimoulas et al., 2011] and so the contract itself will be blamed:

```
expected: Num
given: 'foo'
in: the 1st argument of
    the 1st argument of
    (f: (Num) -> Num) -> res: Num | f (foo) > 10
function foo guarded at line: 2
blaming: the contract of foo
```

### 6.4.2 Parametric Polymorphism

Contracts.js also supports parametric polymorphism for contracts using a technique similar to the work of Guha et al. [Guha et al., 2007].

Parametric polymorphic functions can be defined using `forall`:

```
@ forall <name (,) ...> <contract>
```

Here each `name` is a contract variable to be bound in `<contract>`. For example, the identity function is defined as:

```
@ forall a (a) -> a
function id(x) { return x; }
```

The contract enforces the invariant that for all values, the value applied to `id` will be returned from the function. If the function does not obey this invariant, a contract violation will be triggered. For example, give the following `const5` function:

```
@ forall a (a) -> a
function const5(x) { return 5; }
```

An invocation such as `const5(10)` will throw the error:

```
const5: contract violation
expected: an opaque value
given: 5
in: in the type variable a of
    the return of
    (a) -> a
function const5 guarded at line: 2
blaming: function const5
```

A key idea of parametric polymorphism is that a function cannot inspect the value of a polymorphic type (otherwise it does not really work "forall"). For example, the `inc_if_odd` function behaves like the identity function unless its argument is odd, which violates the parametricity invariant:

```
@ forall a (a) -> a
function inc_if_odd(x) {
    if (x % 2 !== 0) {
        return x + 1;
    }
    return x;
}
```

If we invoke the function with `inc_if_odd(100)`, the following error will be thrown:

```
inc_if_odd: contract violation
expected: value to not be manipulated
given: 'attempted to inspect the value'
in: in the type variable a of
    the 1st argument of
    (a) -> a
function inc_if_odd guarded at line: 2
blaming: function inc_if_odd
```

Note that some operations on values contracts.js cannot guard against (`typeof` in particular) since JavaScript does not provide a mechanism for trapping operations that are performed on primitive values. This could be addressed by using custom operators to rewrite operations like `typeof`. In the next chapter, I will discuss virtual values, a general technique that allows us to do the necessary trapping on primitive values.

Following [Guha et al., 2007], contracts.js also does a basic form of contract inference for polymorphic contracts with first order values. So for example, if a function's argument is specified to be a polymorphic array, contracts.js will check that the array is homogeneous:

```
@ forall a ([...a]) -> [...a]
function arrayId(l) {
    return l;
}
arrayId([1, 2, "three"]);
```

Contracts.js infers that the `a` should be a `Num` for this application of `arrayId` and then throws and error when it discovers `"three"`:

```
arrayId: contract violation
expected: (x) => typeof x === 'number'
given: 'three'
in: in the type variable a of
    the 2nd field of
    the 1st argument of
    ([....a]) -> [....a]
function foo guarded at line: 2
blaming: (calling context for arrayId)
```

Contract inference is done with simple `typeof` checks, so it can only infer the base types.

### 6.4.3 Asynchronous Contracts

Many applications suffer from bugs in the *temporal* behavior of a program execution; a temporal bug occurs when program events happen in the the wrong order (e.g., attempting to free the same memory location twice). *Temporal contracts* [Disney et al., 2011] provide a general framework to enforce these temporal properties via a contract system by specifying the order in which events are allowed during a program run.

Core to JavaScript's notion of temporality is the event loop. Unlike in a preemptive multithreading language like Java where the scheduler can switch control between threads at any point, programs in JavaScript process each event one after another. Each event is guaranteed to run to completion before returning control to the event loop, which then processes the next event in a queue. While the run-to-completion semantics of JavaScript is easier to reason about than threads, there is still plenty of room for surprising temporal bugs to bite.

One area temporal bugs can arise is when confusing *synchronous* and *asynchronous* functions. A synchronous function is a function that is called before returning control to the event loop whereas an asynchronous function is called on some later turn of the event loop.

As an example of a temporal bug that confuses synchronous and asynchronous functions consider the following API for a Node.js program that provides a caching layer in front of file access (adapted from an example in “Effective JavaScript” [Herman, 2012]):

```
var readFile = require("fs").readFile;
var cache = new Map();

function readCaching(fileName, onsuccess) {
    if (cache.has(fileName)) {
        onsuccess(cache.get(fileName));
    } else {
        readFile(fileName, "utf8", function(err, data) {
            cache.set(fileName, data);
            onsuccess(data);
        });
    }
}
```

As its name suggests, the Node.js function `readFile` reads a file and asynchronously invokes its callback on some later turn of the event loop (once the file has been read from the disk). At first glance `readCaching` seems fine, it calls the `onsuccess` callback on a cache hit otherwise it first calls `readFile` before invoking `onsuccess` once the file reading operation completes.

The problem here is that `readCaching` implements an inconsistent API; sometimes the `onsuccess` handler is called asynchronously (when there is a cache miss) and sometimes synchronously (when there is a cache hit). Client code that is unaware of this inconsistency and expects the `onsuccess` to always be called asynchronously can have its assumptions violated leading to subtle bugs. Consider:

```
var obj = {};
readCaching("foo.txt", function(data) {
    obj.totalLength += data.length;
});
obj.totalLength = 0;
```

If `"foo.txt"` is not in the cache then this snippet works fine since the client has a chance to initialize `obj` before the handler is called. If, on the other hand, `"foo.txt"` is actually in the cache then the handler is called before the client code has a chance to finish initializing `obj`, which means that the final value of `totalLength` will be `NaN` (since in JavaScript `undefined + data.length` will be evaluate to `NaN`). Since the bug depends on what is in the cache, we have a source of nondeterminism that makes reproducing the failure difficult.

To address this problematic temporal behavior, contracts.js supports *async* contracts. We can then rewrite our problematic example by wrapping the function `readCaching` in the contract `(Str, (Str) --> ()) -> ()`.

This contract says `readCaching` is wrapped in a function contract (written `->`) that takes two arguments, a string (`Str`) and an async contract (`(Str) --> ()`) that takes a string and returns undefined. A function wrapped in the standard `->` contract can be invoked either synchronously or asynchronous whereas a function wrapped in the `-->` contract *must* be invoked asynchronously (i.e., on some later turn of the event loop). Since `readCaching` does not obey this specification, when `onsuccess` is synchronously invoked on a cache hit the contract will throw an error blaming `readCaching` for violating its contract.

To implement async contracts, we need a way to know on what turn of the event loop an event is currently executing. A simple way to accomplish this is to make a unique identifier for each event in the loop available to an async contract. Then the process of checking for async/sync behavior can proceed as follows:

- Wrap the async function in its contract

- Record the event loop id in which the wrapping took place
- When the wrapped async function is invoked:
    - If the current loop id is equal to recorded loop id then raise blame
    - Otherwise continue execution

An example implementation of async contracts for just asynchronous checking (ignoring the domain and range contracts for simplicity) would look something like this:

```
function async(f) {
    var loopId = getLoopId();
    return function() {
        if (getLoopId() === loopId) {
            throw new Blame("Called synchronously");
        }
        // invoke the function normally
        return f.apply(this, arguments);
    };
}
```

While the function `getLoopId()` does not exist in JavaScript most JavaScript environments provide the means for us to implement `getLoopId()` ourselves. In particular Node.js provides the function `process.nextTick(cb)` that invokes its callback before the next turn of the event loop. This allows us to implement `getLoopId()` directly; each time `getLoopId` is called the current loop id is returned and `process.nextTick` is used to queue up a callback that increments `loopId` before the next turn of the event loop occurs:

```
var loopId = 0;
function incLoopId() { loopId++; }
function getLoopId() {
    process.nextTick(incLoopId);
    return loopId;
}
```

In browser environments `nextTick` is not available but the `setImmediate` function could be used to a similar effect however it is only available in certain browsers and its standardization is contested. In any event, polyfills for `setImmediate` exist[1] that

[1] https://github.com/YuzuJS/setImmediate

take advantage of clever tricks using `postMessage` (a function meant for cross-document messaging) and web workers.

Unsurprisingly, it is straightforward to implement the dual of an async contract: a sync contract that specifies the function must be invoked on the same turn of the event loop. The only change required is that the loop id when the function is invoked must be the same as when the function was wrapped in the `sync` contract. It is also straightforward to implement a contract that checks that its argument is consistently used either synchronously or asynchronously by checking how it was used the first time and then consistently enforcing the same behavior.

## 6.5 Conclusion

Contracts.js demonstrates how macros can provide a declarative language on top of a runtime library. Runtime libraries like contracts that suffer from poor developer ergonomics can use macros to build powerful and declarative domain specific languages that expand to simple library calls.

# Chapter 7

# Application - Virtual Values

Programming language design is driven by multiple, often conflicting desiderata, such as: expressiveness, simplicity, elegance, performance, correctness, and extensibility, to name just a few.

Macro systems focus on improving *extensibility*: the ability of a programmer using a particular language to extend the functionality and expressiveness of that language. Extensibility is desirable on its own merits; it also helps control language complexity by allowing many aspects of functionality to be delegated to libraries, and it enables grassroots innovation, where individual programmers can extend the language rather than being restricted to particular features chosen by the language designer.

In the last chapter I used macros to extend JavaScript with an additional contract specification language; the contracts.js extension is an example of macros being used to implement a domain specific language. A different avenue for language extension that this chapter investigates is using macros to change the semantics of the existing syntactic forms of a language.

The starting point for this language extension is the observation that language semantics typically involve interaction between code and data, where code performs various *operations* (allocation, assignment, addition, etc.) on data values. The behavior of each operation is typically *hardwired* by the language semantics. If a function wants to perform addition on its argument, then it must be passed a numeric value that can be understood by the built-in addition operation. Consequently, a user-defined `complex` type will not interoperate with code that uses the built-in addition operation.

Computer science has a strong history of virtualizing various well-defined interfaces. For example, virtualizing the interface between a processor and its memory subsystem enabled innovations such as virtual memory, distributed shared memory, and memory mapped files. Virtualizing the entire processor enables multiple virtual machines to run on a single hardware processor, or to migrate between processors.

This chapter explores the benefits of "virtualizing" the interface between code and data values. Specifically, this chapter presents a language extension called *virtual values*. When a primitive operation expects a regular value but finds a virtual value in its place, that operation invokes a *trap* on the virtual value. Each virtual value contains a collection of traps, each of which is a user-defined function that describes how a particular operation should behave on that virtual value.

Virtual values seem to provide a rather useful notion of language extensibility. Of course, validating a language design feature is always difficult. In this chapter, I aim to validate the expressiveness and extensibility benefits of virtual values by illustrating the kinds of language extensions that they enable. These extensions include:

1. Additional numeric types, such as rationals, bignums, complex numbers, or decimal floating points[1], with traditional operator syntax.
2. Units of measure (meters, seconds, etc).
3. Taint tracking.

Each language extension is powerful yet small, thus illustrating that virtual values offer an elegant and expressive mechanism for language extension.

These extensions are nicely composable. The taint extension automatically tracks taint information through all code, including through the complex numbers extension.

To emphasize the modularity benefits of virtual values, consider for a moment the consequences of an alternative architecture in which these extensions are implemented as part of the language itself. This approach radically complicates the language,

[1]Decimal floating point numbers (IEEE 754-2008) avoids the unintuitive rounding errors of binary floating point. This work is partly motivated by discussions within the ECMA TC39 Javascript standardization committee regarding the desire for a decimal floating point library that could support convenient operator syntax.

since each extension may cross-cut the other features and evaluation rules of the language. For example, the taint tracking and complex number extension would interact in a non-trivial fashion, since it would become necessary to track how taint information flows through operations on complex numbers. In contrast, virtual values enable a clear separation of concerns between the various extension modules, and provide a coherent and extensible architecture. Composed virtual values are essentially an instance of the Decorator Pattern [Gamma et al., 1995], which is a fairly general pattern that can be applied to any interface, but it is particularly powerful when applied to the widely-used interface between code and data.

This work is inspired by and builds on top of Miller and Van Cutsem's proposal for JavaScript *Catch-All Proxies* [Miller and Cutsem, Cutsem and Miller, 2010], which provide traps for operations on functions and objects. These object proxies virtualize the interface between code and objects (including function objects). Analogous functionality has been provided in other languages, including via Racket's *chaperones* [Matthew Flatt and PLT, 2010].

Virtual values generalizes these prior ideas to virtualize the interface between code and *all data values*, including primitive values such as integers. This generalization enables additional applications, including the applications from the list above, and may prove helpful for mainstream languages, which typically include a large collection of non-object values.

SmallTalk [Goldberg and Robson, 1983] demonstrated the benefits of pure object-oriented programming, in which all data values are objects, and all operations (including addition and conditional tests) are method calls. SmallTalk supports the definition of proxy objects that implement the `doesNotUnderstand:` method and that delegate to an underlying object, a technique called *behavioral intercession.*

This pure object architecture provides flexibility and partially virtualizes the interface between code and data, since many operations are performed via dynamically-dispatched method calls. Virtual values extends the virtualization provided by pure object languages, and moreover demonstrates that extensibility is not restricted to pure object languages: virtual values enable similar extensibility in languages that are not object oriented, or that are only partially object oriented and which include non-object values.

Language extensibility has been the target of a rich body of prior research. For example, CLOS provides a very flexible *metaobject protocol* [Kiczales et al., 1991], which gives the ability to inspect and modify the behavior of the object runtime system, often in a very general manner. In comparison to CLOS, virtual values provides a focused mechanism for changing the language semantics at a *per-value* granularity, Aspect-oriented programming (AOP) [Kiczales, 1996] focuses on *cross-cutting concerns* that span multiple components of a system. As one example, aspects have been used to enforce fine-grained security policies in browsers [Meyerovich et al., 2010]. Virtual values share similar motivations to AOP, and both enable the developer to insert code at different *point-cuts*, but using virtual values these point-cuts are chosen dynamically (based on where virtual values are used) rather than statically (as in weaving-based approaches to AOP).

In a language with a rich static type system, the "trap dispatch" operations on virtual values could be resolved statically, e.g.via Haskell's [Paul Hudak and Simon Peyton-Jones and Philip Wadler (eds.), 1992] type classes. This static type based approach provides stronger correctness guarantees and improved performance over virtual values, but at a cost of more conceptual complexity and some decrease in flexibility. Overall, virtual values seem best suited to providing extensibility in languages whose static type systems are less rich than Haskell, or in dynamically typed languages. Also, whereas type classes such as Haskell's `Num` class virtualize some language operations (those that manipulate `Num` values), virtual values generalize this idea to all language operations.

The following sections present an overview of the existing proxy support in ES2015 for objects and functions, discuss the implementation of virtual values that extend ES2015 Proxies with support for primitive values, and show a series of languages extensions written using virtual values. An open source implementation of virtual values is publicly available[2].

[2]http://disnetdev.com/sweet-virtual-values/

### 7.0.1 Virtual Values and ES2015 Proxies

As a first step in implementing virtual values in JavaScript we can take advantage of the existing behavioral intercession facility in JavaScript, namely ES2015 Proxies [Miller and Cutsem, Cutsem and Miller, 2010], which, unlike virtual values, are designed to only work for objects and functions.

ES2015 proxies are objects in JavaScript that wrap other objects and dispatch to user-defined functions for every operation performed on the proxy. Proxies are created by calling the `Proxy` constructor with two arguments, the object or function to proxy and a handler object where each property on the handler corresponds to behavior to trap. For each operation performed on the proxy, the corresponding handler method is invoked. For example, the following proxy logs each property access.

```
var p = new Proxy({}, {
    get: function(target, name) {
        console.log("accessing property: " + name);
        return target[name];
    }
});
```

While the full handler API contains 14 methods to address many operations that can be performed on functions and objects in JavaScript, for clarity of presentation this chapter will consider a subset of the proxy API that only includes the `get` and `set` traps (which correspond to property get and sets) along with the `apply` trap (which corresponds to function application).

```
var p = new Proxy(x, {
  // x[name]
  get: function(target, name) { /* ... */ },
  // x[name] = val
  set: function(target, name, val) { /* ... */ },
  // x(foo, bar, baz)
  apply: function(target, thisArg, args) { /* ... */ }
});
```

When a trap is invoked, the first argument provided to each trap, `target`, is a reference to the underlying value being proxied. If a trap is not defined, a proxy will perform the operation on the underlying object directly.

### 7.0.2 Implementing Value Handlers

Before discussing the implementation details of our proxy extension, it is useful to consider the API design we are trying to achieve. In particular, we would like the design to be as familiar to existing users of ES2015 proxies as possible. To that end, the design makes two changes to the existing proxy API. First, a proxy constructor can wrap primitive values in addition to objects and functions (the existing proxy constructor throws an error if called with a primitive). The second change is to add the `unary`, `left`, and `right` traps to the proxy handler corresponding to unary operations on the proxy and binary operations where the proxy is the right or left operand respectively.

```
var p = new Proxy(42, {
    // ‘op‘ p
    unary: function(target, op) {
        // ...
    },
    // p ‘op‘ right
    left: function(target, op, right) {
        // ...
    },
    // left ‘op‘ p
    right: function(target, op, left) {
        // ...
    }
});
```

The design can be implemented on top of the existing proxies via macros. The key issue that macros allows us to resolve is dispatching to the handlers for primitive operations. This dispatching step works by defining sweet.js custom operators for each primitive operation that expands to a call to a function `binary` defined by the rewriting library:

```
operator + 14 left { $l, $r } => #{ binary("+", $l, $r) }
operator - 14 left { $l, $r } => #{ binary("-", $l, $r) }
...
```

At a high level, the `binary` function performs the standard primitive operation if neither of its arguments are proxies and invokes the appropriate handler trap if either

**Figure 7.1: Proxy Constructor**

```
// get a reference to the Proxy constructor
var ESProxy = window.Proxy;
// a WeakMap holds weak references to
// its keys, allowing the GC to reclaim
// them if there are no other references
var unproxy = new WeakMap();

function Proxy(val, handler) {
    var p;
    if (typeof val !== 'object') {
        // since the ESProxy constructor would throw
        // an error for non-objects we need to pass
        // in a dummy object
        p = new ESProxy({}, handler);
    } else {
        p = new ESProxy(val, handler);
    }
    // map the proxy to the handler
    unproxy.set(p, handler);
    return p;
}
```

**Figure 7.2: Binary and Unary Functions**

```
function unary(op, p) {
    if (unproxy.has(p)) {
        return unproxy.get(p).unary(op);
    }

    if (op === "!") return !p;
    // ...
}
function binary(op, left, right) {
    if (unproxy.has(left)) {
        return unproxy.get(left).left(left, op, right);
    }
    if (unproxy.has(right)) {
        return unproxy.get(right).right(right, op, left);
    }

    if (op === "+") return left + right;
    // ...
}
```

argument is a proxy. In order to perform these operations the `binary` function must (1) have the ability to recognize proxies and (2) gain a reference to a proxy handler.

A straightforward way of implementing these functions is for the proxy constructor and `binary` to share an `unproxy` map the associates proxy references with their handler objects. Figure 7.1 shows the implementation of the `Proxy` constructor and Figure 7.2 shows the implementation of the `binary` and `unary` functions.

Because `binary` and `unary` share a reference to `unproxy` with the proxy constructor, `binary` and `unary` are able to recognize every proxy that has been created in the system and dispatch to the appropriate handler.

## 7.1 Language Extension Examples

To illustrate the expressiveness and extensibility benefits of proxies, I now use the virtual values API to implement a series of interesting language extensions. Each extension is small yet adds significant expressive power to the language.

### 7.1.1 Identity Proxy

As a starting point for the series of language extensions, Figure 7.3 sketches a simple proxy that has no effect on program evaluation. In particular, evaluating `makeIdentityProxy(x)` returns a proxy in which each trap handler simply performs the appropriate operation on the underlying argument `x`. For unary operations, the `unary` trap dispatches to an auxiliary object `unaryOps`, which maps each unary operator string to a function that performs the corresponding operation. The `left` and `right` traps similarly dispatch to the `binOps` lookup table.

### 7.1.2 Tainting Extensions

Several languages, such as Perl, provide tainting as a built-in feature of the language implementation, which introduces additional complexity into the compiler/interpreter and runtime data representations.

Proxies allow this complexity to be isolated into a small extension module, as shown in Figures 7.4 and 7.5. The function `taint` takes an argument `x` and returns

**Figure 7.3: Identity Proxy**

```
function makeIdentityProxy(x) {
    return new Proxy(x, {
        unary: function(target, op) { return unaryOps[op](target) },
        left: function(target, op, r) { return binaryOps[op](target, r) },
        right: function(target, op, l) { return binaryOps[op](l, target) },
        get: function(target, name) { return target[name] },
        set: function(target, name, val) { return target[name] = val },
        apply: function(target, thisArg, argsList) {
            return target.apply(thisArg, argsList)
        }
    });
}

var unaryOps = {
    "-": function(x) { return -x },
    "!": function(x) { return !x },
    // etc. for all unary ops
};
var binaryOps = {
    "+": function(x, y) { return x + y },
    "-": function(x, y) { return x - y },
    // etc. for all binary ops
};
```

**Figure 7.4: Tainting Proxy Pt. 1**

```
var unproxy = new WeakMap();
function taint(x) {
    if (isTainted(x)) { return x; }
    var handler = {
        // store the original untainted value for later
        originalValue: x,
        unary: function(target, op, operand) {
            return taint(unaryOps[op](target));
        },
        left: function(target, op, right) {
            return taint(binaryOps[op](target, right));
        },
        right: function(target, op, left) {
            return taint(binaryOps[op](left, target));
        },
        get: function(target, name) {
            return taint(target[name]);
        },
        set: function(target, name, val) {
            return target[name] = taint(val);
        },
        apply: function(target, thisArg, argsList) {
            return taint(target.apply(thisArg, argsList));
        }
    };
    var p = new Proxy(x, handler);
    unproxy.set(p, handler)
    return p;
}
```

**Figure 7.5: Tainting Proxy Pt. 2**

```javascript
function isTainted(x) {
    // a value is tainted if it's in the unproxy map
    if (unproxy.has(x)) {
        return true;
    }
    return false;
}

function untaint(value) {
    if (isTainted(value)) {
        // pulls the value out of its tainting proxy
        return unproxy.get(value).originalValue;
    }
    return value;
}
```

a proxy that behaves much like `x`, in that all traps first perform the corresponding operation on `x` but then taint the result. Each tainting proxy is stored in an `unproxy` map, so that tainted proxies can be recognized by the `untaint` and `isTainted` functions. A value is *tainted* if it is one of these tainting proxies and is *untainted* otherwise. To untaint values (after they have been appropriately sanitized), the handler record in the tainting proxy keeps the original value in the `originalValue` field, so that it can be later returned by `untaint`.

Based on these definitions, tainted values now propagate through all primitive operations of the language. For example, `4 + taint(5)` evaluates to a tainted 9, that is, a tainting proxy whose underlying value is 9.

### 7.1.3 Complex Numbers

An often-requested feature of a programming language is the ability to introduce additional numeric types beyond what are provided in the language implementation, and to manipulate these additional types using traditional operator syntax. In many languages, this kind of extension is difficult. For example, Java provides `Bignums`, but only as a library with awkward method invocation syntax, and it does not provide rationals, complex numbers, or decimal floating points.

Figures 7.6 and 7.7 illustrates how to add additional numeric type, namely complex numbers. The function `makeComplex` takes as input the two components of a complex number, and creates a proxy that dispatches unary and binary operations appropriately. For binary operations, the `left` trap first checks if the right argument `right` is a ordinary number or a complex number. If `right` is complex, then we pass its real and imaginary components (extracted from `right`'s handler) to the appropriate function in the `complexBinOps` table. Otherwise we assume that `right` is a real number and pass 0 as the imaginary component to the `complexBinOps` table function. The `right` trap is simpler, since its left argument is never complex.

The example implementation also defines the variable `i`, from which client code can conveniently construct arbitrary complex numbers, for example

```
1.0 + (1.0 * i)
```

As mentioned in the introduction, virtual values enable *compositional* language

**Figure 7.6: Complex Numbers Proxy Pt. 1**

```
var unproxy = new WeakMap();
function makeComplex(real, img) {
    var handler = {
        real: real,
        img: img,
        unary: function(target, op) {
            return complexUnaryOps[op](real, img);
        },

        left: function(target, op, right) {
            var rhandler = unproxy(right, key);
            if (rhandler) {
                return complexBinOps[op](real, img, rhandler.real, rhandler.img);
            } else {
                return binaryOps[op](real, img, right, 0);
            }
        },
        right: function(target, op, left) {
            return binaryOps[op](left, 0, real, img);
        }
    };
    var p = new Proxy({}, handler);
    unproxy.set(p, handler);
    return p;
}
```

**Figure 7.7: Complex Numbers Proxy Pt. 2**

```
var complexUnaryOps = {
    "-": function(real, img) {
        return makeComplex(-real, -img);
    }
    // ...
};

var complexBinOps = {
    "+": function(lReal, lImg, rReal, rImg) {
        return makeComplex(lReal + rReal, lImg + rImg);
    },
    // ...
};

function isComplex(value) {
    if(unproxy.has(value)) {
        return true;
    }
    return false;
}

var i = makeComplex(0, 1);
```

extension. For example, both the complex number extension and the tainting extension can be used:

```
var n = taint(1.0 + (1.0 * i));
if (isTainted(42 * n)) {
  // handle a tainted complex number
}
```

Note that proxies are not a "silver bullet" for compositionality. In particular, proxies use a double dispatch protocol for overloading binary operators. Consequently, two independent proxy-based extensions, say `Complex` and `Rational`, may not be composable, since neither implementation knows how to add a complex and a rational number. Generic functions, as in CLOS [Kiczales et al., 1991] and elsewhere, provide more flexibility but with some additional complexity.

### 7.1.4 Dynamic Units of Measure

Several type systems (see for example, [Kennedy, 1997]) have been proposed to track *units of measure*, such as meters or seconds, and to avoid the confusion of units that caused the Mars Climate Orbiter mishap [Stephenson et al., 1999]. We use the term *quantity* to mean a floating point number annotated with zero or more units of measure, each of which may have an associated integer *multiplicity* or index (as in second$^{-2}$). Thus, an example quantity is 9.81 meters second$^{-2}$.

Proxies provide a convenient means to track units dynamically, as illustrated in Figures 7.8 and 7.9[3]. Each quantity is represented as a chain of proxies, terminating in a floating point number. Each proxy handler contains a unit of measure, an integer index, and an underlying value (the next proxy in the chain, or a floating point number). The function `makeQuantity` creates these proxies, ensuring that each proxy has a non-zero index, and that the proxy chain is kept in lexicographic ordering of units with at most one proxy for each unit (i.e., no duplicates).

Unary and binary operators on a quantity propagate down the proxy chain to the underlying numbers, provided the units are appropriately compatible. In particular, `"+"` requires that its arguments have identical units by calling the function

[3]For simplicity, this implementation does not support *dimensions* (such as mass), but only units of measure (such as kilograms).

**Figure 7.8: Units Proxy Pt. 1**

```
var unproxy = new WeakMap();
function makeQuantity(unitName, index, value) {
    var handler = unproxy.get(value);
    if (index === 0)  return n;
    if (handler) {
        if (handler.unit === unitName) { // same unit avoid duplicates
            return makeQuantity(unitName, handler.index + index, handler.value);
        } else if (handler.unit > unitName) { // keep the proxies ordered
            return makeQuantity(handler.unit, handler.index,
                                    makeQuantity(unitName, index, handler.value));
        }
    }
    var handler = {
        unit: unitName,
        index: index,
        value: value,
        unary: function(target, op) {
            return unaryOps[op](unitName, index, value);
        },
        left: function(target, op, right) {
            return leftOps[op](unitName, index, value, right);
        },
        right: function(target, op, left) {
            return rightOps[op](unitName, index, value, left);
        }
    };
    var p = new Proxy({}, handler);
    unproxy.set(p, handler);
    return p
}
```

**Figure 7.9: Units Proxy Pt. 2**

```
var unaryOps = {
    "-": function(unitName, index, value) {
        return makeQuantity(unitName, index, -value);
    },
    // ...
};
var leftOps = {
    "+": function(unit, index, value, right) {
        return makeQuantity(unit, index, (value + dropUnit(unit, index, right)));
    },
    // ...
};
var rightOps = {
    "+": function(unit, index, value, left) {
        throw new Error("Incompatible types");
    },
    // ...
};
function dropUnit(unit, index, value) {
    var handler = unproxy.get(value);
    if(handler) {
        if (handler.unit === unit && handler.index === index) {
            return handler.value;
        }
    }
}
function makeUnit(unit) {
    return makeQuantity(unit, 1, 1);
}
```

`dropUnit(unit, index, right)`, which ensures that the right argument `right` has the unit `unit` with index `index`, and returns the unwrapped version of `right`. The `makeUnit` function can be used by client code to create desired units of measure, as in:

```
var meter = makeUnit("meter");
var second = makeUnit("second");
var g = 9.81 * meter / second / second;
g + 1 // dynamic unit mismatch error
```

### 7.1.5 Handling `test`, `geti`, and `seti`

In prior work [Austin et al., 2011] we presented a more general form of the virtual values in this chapter that included three more traps not shown here: `test`, `geti`, and `seti`.

The `test` trap corresponded to a proxy being used in a conditional (e.g., the operation `if (p) { /* ... */ }` would invoke the `test` trap on `p`'s handler). Implementing `test` is relatively straightforward since we can use macros to rewrite `if` statements just like we were able to rewrite the primitive operators:

```
let if = macro {
    rule { ( $cond ... ) { $body ... } } => {
        if (test($cond ...)) {
            $body ...
        }
    }
}
```

The `test` function can then dispatch to the proxy handlers `test` trap:

```
function test(cond) {
    if (unproxy.has(cond)) {
        return unproxy.get(cond).test(cond);
    }
    return cond;
}
```

The `geti` and `seti` are more interesting and present a challenge to implement and highlight an important area of future work for sweet.js.

These traps correspond to their proxy being used in the index of an object get or set. For example, if `p` is a proxy and `o` is any normal object, then the operation `o[p]` would invoke the `geti` trap on `p`.

While the other traps can be implemented by either custom operators or a simple macro to override a standard JavaScript form like `if`, in order to implement the `geti/seti` traps we would need some way to transform syntax of the form `obj[p]` into `geti(p, obj)`, which is not currently possible.

While we cannot currently do this kind of expansion, sweet.js could take inspiration from Racket's design of *implicit forms*[4]. Implicit forms in Racket are syntax forms like `#%app`, which corresponds to function calls. In Racket the s-expression `(foo a b)` is transformed into the implicit form `(#%app foo a b)` before the function call actually takes place. Racket's module system allows a programmer to install a syntax transformer associated with a particular implicit form, changing them different meaning of, for example, function application within a particular module.

Internally sweet.js already uses an analogue of implicit forms in the terms produced from enforestation. A computed object lookup (e.g., `o[p]`) is enforested to a corresponding term. In the future, sweet.js could add the ability to manipulate enforested terms in a similar way to Racket's implicit forms.

Since we cannot currently implement the `geti/seti` traps, what does that mean for the virtual values extension from the previous sections? If we just use the design described so far it means that if a virtual value created from our rewriting approach is used as a key in a property get or set it will fail:

```
var p = new Proxy(1, { /*..*/ });
var l = [1,2,3];
l[p] // undefined
```

This happens because `p` is still being represented as an object and property lookup uses the string representation of `p` (by default `"[object Object]"`). We can address this for primitive values by changing the proxy constructor to include a `toString` method on the dummy object we proxy:

```
// revised proxy constructor
function Proxy(val, handler) {
    var p;
```

[4]http://docs.racket-lang.org/guide/module-languages.html#%28part._implicit-forms%29

```
    if (typeof val !== 'object') {
        p = new ESProxy({
            // return the string representation of the
            // original value
            toString: function() { return val + ""; }
        }, handler);
    } else {
        p = new ESProxy(val, handler);
    }
    unproxy.set(p, handler);
    return p;
}
```

The JavaScript environment will call the `toString` method when the proxy is used in a property lookup.

This approach handles the immediate problem of property lookup when proxying a primitive value but it does not give the proxy control over how to represent the string. We can address this by adding another trap in the handler `stringRepr`:

```
// revised proxy constructor
function Proxy(val, handler) {
    var p;
    if (typeof val !== 'object') {
        p = new ESProxy({
            // return the string representation of the
            // original value
            toString: function() { return handler.stringRepr(val); }
        }, handler);
    } else {
        p = new ESProxy(val, handler);
    }
    unproxy.set(p, handler);
    return p;
}
```

This change allows us to cleanly implement the identity and lazy evaluation extensions. Complex numbers and dynamic units of measure are not affected since they do not make sense as a key to an object.

The extension for which this loss in functionality presents an issue is tainting. If a value `t` is a tainted proxy, then the operation `o[t]` should result in another tainted value but this taint propagation is not possible without the `geti` and `seti` traps.

## 7.2 Principles of Design

Bracha and Ungar propose three design principles for reflective APIs [Bracha and Ungar, 2004], namely encapsulation, stratification, and ontological correspondence.

Proxies satisfy the *principle of encapsulation*, since the proxy API does not expose details regarding the underlying implementation of the language.

Proxies also satisfy the *principle of stratification*, since there is a clear distinction between base level values (both raw values and proxies), and meta-level values (the handler for a proxy value). In particular, there is no way for a user of a proxy value to access the underlying handler. Evaluating `(proxy a h)["unary"]` does not return the `unary` trap function of the handler `a`; instead it invokes `a`'s `get` trap on the argument "unary".

Finally, proxies satisfy the *principle of ontological correspondence*, since each trap handler corresponds directly to a particular operation being performed by code on a (virtual) data value.

## 7.3 Conclusion

The language extensions presented in Section 7.1 provide anecdotal evidence that virtual values provide a flexible and useful language extension mechanism. In addition, it is fairly straightforward to implement virtual values using sweet.js macros.

Note that while actually implementing virtual values in a JavaScript engine such as SpiderMonkey or V8 rather than by macros would be "better" in the sense that it would be more performant and available to more users of JavaScript, this would force us to invest considerable engineering effort and time, since modern JavaScript engines are tuned for performance rather than extensibility, before the design has been sufficiently understood. The macro implementation technique allows rapid design prototyping.

Virtual values are motivated by the rich proliferation of research on various kinds of wrappers and proxies, including higher-order contracts [Findler and Felleisen, 2002, Findler and Blume, 2006], language interoperation via proxies [Gray et al., 2005], and hybrid and gradual typing [Siek and Taha, 2007, Flanagan, 2006] and space-efficient gradual typing [Siek and Wadler, 2010]. It is possible that virtual values may allow some of this research to be performed by experimenting within a language with virtual values, rather than by designing new languages and implementations.

# Chapter 8

# Design and Related Work

Macro systems have many (often competing) design goals. For example, we might want our macro system to be easy to learn but also be expressive enough to perform every desired syntactic transformation, making the system harder to learn. A list of potential design goals for macro systems include:

- Composability
- Correctness
- Expressiveness
- Declarativity
- Easy to learn
- Easy to reason about
- Full syntactic abstraction

The design goals listed here do not have a precise definition, rather they are suggestive along a spectrum. For example, consider composability, which means that two macros written by independent authors will work correctly when used in the same program. Composability is not a binary condition, rather it is a suggestive metric that is used in comparison to other systems. For example, the C [Harbison and Steele, 1984] preprocessor (CPP) provides macros that are not hygienic so two CPP macros can clash if they both inadvertently use the same names for bindings. In contrast, Scheme and sweet.js macros are hygienic so this kind of composability failure will not occur.

While the design of sweet.js has been guided by many goals, three stand out in particular: declarative macro definitions, composability, and full syntactic abstraction.

## 8.1 Declarative Definitions

The goal of providing declarative definitions has driven the design of the pattern language for macro declarations along with the use of enforest to provide pattern classes. In Scheme, declarative definitions has a long history going back to the macro-by-example [Kohlbecker and Wand, 1987] style macros where rather than write procedures to manipulate s-expressions directly macro authors could write declarative patterns.

Sweet.js, following Honu [Rafkind and Flatt, 2012], extends the macro-by-example design by using enforestation to provide pattern classes that allow macro authors to pattern match on grammar productions. Pattern classes free macro authors from reasoning about how to match different kinds of syntax and allows them to focus on the structure of their macros.

As described in Chapter 2, sweet.js also provides the ability for macro authors to define custom pattern classes via the `invoke` mechanism. The `invoke` mechanism allows programmers to extend the pattern language in a declarative manner.

The `invoke` mechanism is a simplified version of Racket's `syntax-parse` [Culpepper and Felleisen, 2010]. The `syntax-parse` form in Racket allows macro authors to create robust declarative patterns with expressive error handling. In sweet.js, `invoke` currently provides macro authors the ability to extend the pattern language, but without the more powerful error handling features of `syntax-parse`.

## 8.2 Composability

The design goal of composability provides the ability for macros written by independent authors to coexist. Composable macros are macros which work correctly when used within another macro.

There are two primary tools that sweet.js uses to achieve composability: hygiene, which provides composable bindings and was discussed in Chapter 5, and expansion order, which give control to the "correct" macro.

Expansion order is a point often left implicit in discussions of macro systems. By expansion order I mean the order in which macros are invoked when a series of macros are nested. Consider:

```
m1 {
    m2 {
        m3 {
            ...
        }
    }
}
```

There are two possible choices, either outside-in (`m1` is invoked first) or inside-out (`m3` is invoked first). Both Racket and sweet.js choose outside-in. This choice allows the outer macro to manipulate nested macros before they are expanded.

While this choice is important, both Racket and sweet.js have tools to invert expansion order when necessary. Racket has the function `local-expand` (formalized in [Flatt et al., 2012]) which allows a `syntax-case` macro to force expansion on its subforms[1]. In sweet.js, when a macro uses a pattern class, such as `:expr` to match an expression, enforestation essentially does a `local-expand` until an expression can be matched.

## 8.3 Full Syntactic Abstraction

The full syntactic abstraction design goal is what allows sweet.js to build out syntactic features from future versions of JavaScript like the ES2015 macros described in Chapter 2. The goal here is to allow macro authors to build any syntactic form as a macro; a user of a sweet.js macro should not have to know that a given form is implemented as a macro.

This goal is not shared by all macro systems. For example, invocations of a macro in both template Haskell [Sheard and Jones, 2002] (via the syntax `$(...)`) and Rust [Matsakis and Klock, 2014] (via the syntax `macroName!`) are distinguished from other syntactic forms. This difference demonstrates a divergent design philosophy, one

[1]sweet.js also provides preliminary support for `local-expand`

in which macros are seen as an aid to address syntactic repetition (Template Haskell and Rust) and the other as a tool for language evolution (Racket and sweet.js).

Beyond not distinguishing syntactically between macro forms and builtin forms, Racket and sweet.js differ primarily in the kinds of syntactic forms that must be handled by the macro system. To a close approximation, Racket has a single syntactic form (an s-expression) whereas sweet.js must address many different forms (syntax with a prefix name such as `if` or `for`, operators with precedence like `+` etc.).

### 8.3.1 Infix Macros

Infix macros were motivated by the ES2015 arrow function use-case. While powerful and necessary to successfully enable macros that could implement the new ES2015 forms, infix macros are complicated to reason about because of their asymmetric pattern matching. In my formalism only full terms in the prefix can be matched against though the actual implementation loosens this restriction somewhat for certain special cases. It is an open question if a different design could lead to a more reasonable infix macro implementation.

### 8.3.2 Implicit forms

An often requested feature[2] is to allow macros to match against newlines and other whitespace. This feature is motivated by the desire to implement syntax forms with significant whitespace inspired by languages like CoffeeScript and Python.

Overloading whitespace is not a new idea [Stroustrup, 1998]. A promising avenue for sweet.js to pursue is the approach taken by Racket and Fortress [Allen et al., 2005], which use *implicit forms* to represent, for example, the juxtaposition of two identifiers. Macros could then be extended to take advantage and override the behavior of implicit forms. In addition to whitespace, implicit forms could also be used to implement the `geti` and `seti` traps for virtual values by allowing macro authors to override the computed property (`foo[bar]`) form.

[2]e.g., https://github.com/mozilla/sweet.js/issues/55

## 8.4 Discussion

The experience of sweet.js suggests two considerations in the design of macro systems for languages with complex grammars. One consideration is how languages can and should handle lexical ambiguities, and the other is how to approach the goal of full syntactic abstraction.

Note that lexical ambiguities are not unique to JavaScript. The grammar of C and C++, for example, distinguishes two kinds of lexical identifiers: variable names and type names. This distinction cannot be resolved lexically, and C/C++ parsers requires additional context to disambiguate [McKeeman, 1991]. Perl shares the same `/` ambiguity with JavaScript [PerlMonks].

Since lexical ambiguities often arise in complex grammars, the experience of sweet.js raises two points worth considering. Firstly, for language designers building new languages, it is worth keeping in mind that introducing lexical ambiguities involves additional tradeoffs than simply the design of the lexer and parser for that language. In particular, allowing ambiguities to creep into the grammar will potentially influence tools, such as a macro system, that work over the lexical structure in isolation (i.e., without the context provided by the parser defined over a fixed grammar). Secondly, for languages that already have lexical ambiguities, the experience of sweet.js suggests that it might still be possible to resolve those ambiguities. Resolving ambiguities involves a fair amount of work to determine a correct prefix, but since this dissertation has shown it can be done for JavaScript it seems likely that it can also be done for other languages as well.

Beyond lexical ambiguities, an additional consideration for macro systems is how to address full syntactic abstraction. Note that not all macro systems need to or will be designed with the goal full syntactic abstraction. For example, both Template Haskell and Rust have intentionally chosen to build their macro systems in ways that are counter to the goal of full syntactic abstraction.

However, for macro systems that want to achieve some form of full syntactic abstraction, the experience of sweet.js suggests a few areas that are worth exploring further. In particular, the enforestation technique of Honu [Rafkind and Flatt, 2012] is powerful. It enables declarative pattern matching along with the ability to override and define new operators.

Beyond the basic enforestation algorithm the experience of sweet.js also suggests the need to support more flexible syntactic forms. For a language like Racket where all syntactic forms are prefixed by a distinguishing identifier, prefix macros are sufficient. However, in languages with more complex grammars, not all syntactic forms contain distinguishing prefixes. For example, in JavaScript arrow functions cannot be implemented with a prefix macro. Sweet.js provides support for these more flexible syntactic forms via infix macros.

Finally, the experience of building virtual values with sweet.js macros illuminated the need for a feature like implicit forms. Implicit forms allow a macro system to override the meaning of syntactic forms that do not even contain a distinguishing identifier.

## 8.5 Related Work

Syntactic abstraction facilities have long been a part of the design of programming languages. Some systems, such as the C preprocessor [Harbison and Steele, 1984] and LaTeX [Lamport, 1994], work over just flat token streams.

More powerful systems require additional structure and can be roughly broken down into two groups—those that manipulate semi-structured data such as s-expressions in Lisp and Scheme [Clinger, 1991, Dybvig et al., 1992] and those that manipulate ASTs [Weise and Crew, 1993] or provide hooks to extend the grammar [Cardelli et al., 1994].

### 8.5.1 Macro Systems

Macros have been extensively used and studied in Lisp [Foderaro et al., 1983, Pitman, 1980] and related languages for many years. Scheme in particular has embraced macros, pioneering the development of declarative definitions [Kohlbecker and Wand, 1987] and working out the hygiene conditions for term rewriting macros (rule macros) [Clinger, 1991] and procedural macros (case macros) [Dybvig et al., 1992] that enable true composability. In addition there has been work to integrate procedural macros and module systems [Flatt, 2002, Ghuloum and Dybvig, 2007]. Racket takes it even further by extending the Scheme macro system with deep hooks into the compilation process [Tobin-Hochstadt et al., 2011, Flatt et al., 2012] and robust pattern specifications [Culpepper and Felleisen, 2010].

In addition, there are a number of macro systems for languages with more traditional syntax that are not fully delimited. As mentioned before, sweet.js is most closely related to Honu [Rafkind and Flatt, 2012, Rafkind, 2013]. In contrast with Honu, which does not include regular expression literals, sweet.js solves the reader ambiguity problem for JavaScript and introduce infix macros.

Macro systems that use a similar technique as Honu include Fortress [Allen et al., 2009] and Dylan [Bachrach et al., 1999], but they only provide support for term rewriting macros (our `rule` macros). Dylan's auxiliary rules are similar to our `invoke` pattern class. Nemerle [Skalski et al., 2004] also uses a similar technique but does not allow local definitions of macros.

The Marco system [Lee et al., 2012] is an interesting alternative approach that presents a cross-language macro system. Rather than tightly integrate the macro system with a specific language, Marco provides a separate macro definition language that can compile to multiple languages. While this approach provides generality it sacrifices language specific expressiveness (e.g., name clashes are errors in their system while they are just renamed in sweet.js).

Recently work has even begun on formalizing the hygiene condition for Scheme [Adams, 2015]. Prior presentations of hygiene have either been operational [Dybvig et al., 1992] or restricted to a typed subset of Scheme that does not include `syntax-case` [Herman, 2010, Herman and Wand, 2008].

### 8.5.2 Extensible Syntax

There are a variety of systems that provide syntactic extension via techniques related to extensible grammars [Cardelli et al., 1994]. Some systems such as SugarJ [Erdweg et al., 2011], OMeta [Warth and Piumarta, 2007], Xtc [Grimm, 2006], Xoc [Cox et al., 2008], and Polyglot [Nystrom et al., 2003] provide extensible grammars but require the programmer to reason about parser details. Multi stage systems such as Mython [Riehl, 2009] and MetaML [Taha and Sheard, 1997, Sheard et al., 2000] can also be used to create macros systems like MacroML [Ganz et al., 2001]. Systems like Stratego [Visser, 2003] transforms syntax using its own language, separate from the host language. Metaborg [Bravenboer and Visser, 2004] and SugarJ [Erdweg et al., 2011] are syntax extension facilities built on top of Stratego.

To the best of my knowledge the only system besides sweet.js that provides syntactic abstraction facilities in JavaScript is ExJS [Wakita et al., 2014]. ExJS is a hygienic macro system for JavaScript based on a staged parsing architecture (rather than a more direct manipulation of syntax as in sweet.js). ExJS supports pattern macros (similar to `rule` macros in sweet.js) but the staged parsing approach does not enable the more general procedural macros (the `case` macros in sweet.js).

### 8.5.3 Template Meta-Programming

C++ templates [Alexandrescu, 2001] are a powerful compile time meta programming facility for C++. In contrast to C++ templates sweet.js provides more syntactic flexibility in the definable syntactic forms along with the ability to define transformations in the host language rather than the just the template language.

Template Haskell [Sheard and Jones, 2002] makes a tradeoff by forcing the macro call sites to always be demarcated. The means that macros are always a second class citizen; macros in Haskell cannot seamlessly build a language on top of Haskell in the same way that Racket and sweet.js can.

## 8.6 Conclusion

This dissertation has presented sweet.js, a hygienic macro system for JavaScript. Sweet.js includes a novel reader for JavaScript that correctly separates the lexing and parsing phases of JavaScript. Sweet.js extends the enforestation technique from Rafkind [Rafkind and Flatt, 2012, Rafkind, 2013] to address full JavaScript and includes infix macros that enable the implementation of syntactic forms in ES2015.

This dissertation has presented a formalization of macro expansion that includes a treatment of hygiene adapted from Flatt [Flatt, 2015] for the JavaScript setting. In addition, I have demonstrated the effectiveness of sweet.js by implementing two kinds of applications: (1) a contract system that shows the ability of sweet.js to implement a domain specific language and (2) virtual values that show how sweet.js can be used for a deeper kind of language extension. These applications of sweet.js also identify and illustrate areas of future work for sweet.js or any macro system that attempts to provide full syntactic abstraction.

# Bibliography

M. D. Adams. Towards the Essence of Hygiene. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015*, 2015.

A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, Apr. 2001.

E. Allen, D. Chase, J. Hallett, and V. Luchangco. The Fortress Language Specification. 2005.

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

T. H. Austin, T. Disney, and C. Flanagan. Virtual Values for Language Extension. In *OOPSLA '11: Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, Oct. 2011.

J. Bachrach, K. Playford, and C. Street. D-expressions: Lisp power, Dylan style. *Style DeKalb IL*, 1999.

E. Barzilay, R. Culpepper, and M. Flatt. Keeping It Clean with Syntax Parameters. In *Proceedings of the Workshop on Scheme and Functional Programming*, 2011.

A. Bawden and J. Rees. Syntactic Closures. In *Proceedings of the 1988 ACM conference on LISP and functional programming - LFP '88*, 1988.

G. Bracha and D. Ungar. Mirrors: Design Principles for Meta-Level Facilities of Object-Oriented Programming Languages. In *OOPSLA*, 2004.

M. Bravenboer and E. Visser. Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation Without Restrictions. *OOPSLA*, 2004.

L. Cardelli, F. Matthes, and M. Abadi. Extensible Syntax with Lexical Scoping. 1994.

M. Carrillo, J. G. Molina, E. Pimentel, and I. Repiso. Design by Contract in Smalltalk. *Journal of Object Oriented Programming*, 1996.

W. Clinger. Macros That Work. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages - POPL '91*, 1991.

R. Cox, T. Bergan, A. T. Clements, F. Kaashoek, and E. Kohler. Xoc, An Extension-Oriented Compiler for Systems Programming. In *ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, 2008.

R. Culpepper and M. Felleisen. Fortifying Macros. *ICFP*, 2010.

T. V. Cutsem and M. S. Miller. Proxies: Design Principles for Robust Object-oriented Intercession APIs. In *Dynamic Languages Symposium*, 2010.

C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct Blame for Contracts: No More Scapegoating. *POPL*, 2011.

T. Disney, C. Flanagan, and J. McCarthy. Temporal Higher-Order Contracts. In *ICFP '11: Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, 2011.

R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic Abstraction in Scheme. *Lisp and Symbolic Computation*, 1992.

S. Erdweg, L. C. L. Kats, T. Rendel, C. Kästner, K. Ostermann, and E. Visser. SugarJ: Library-Based Language Extensibility. In *OOPSLA '11: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, 2011.

N. Faubion. The sparkler project. URL `https://github.com/natefaubion/sparkler`.

R. B. Findler and M. Blume. Contracts as Pairs of Projections. In *International Symposium on Functional and Logic Programming*. 2006.

R. B. Findler and M. Felleisen. Contracts for Higher-Order Functions. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, 2002.

C. Flanagan. Hybrid Type Checking. In *Symposium on Principles of Programming Languages*, 2006.

M. Flatt. Composable and Compilable Macros: You Want It When? In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, 2002.

M. Flatt. Binding as Sets of Scopes: Notes on a new model of macro expansion for Racket, 2015. URL `http://www.cs.utah.edu/~mflatt/scope-sets-5/`.

M. Flatt, R. Culpepper, D. Darais, and R. B. Findler. Macros that Work Together: Compile-Time Bindings, Partial Expansion, and Definition Contexts. *Journal of Functional Programming*, 2012.

J. K. Foderaro, K. L. Sklower, and K. Layer. *The FRANZ Lisp Manual.* University of California Berkeley, California, 1983.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, Boston, MA, 1995. ISBN 0201633612.

S. E. Ganz, A. Sabry, and W. Taha. Macros as Multi-Stage Computations: Type-Safe, Generative, Binding Macros in MacroML. *ICFP*, 2001.

A. Ghuloum and R. K. Dybvig. Implicit Phasing for R6RS Libraries. *ICFP '07 Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, 2007.

A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, 1983. ISBN 0-201-11371-6.

K. E. Gray, R. B. Findler, and M. Flatt. Fine-Grained Interoperability Through Mirrors and Contracts. In *OOPSLA*, 2005.

R. Grimm. Better Extensibility Through Modular Syntax. *PLDI*, 2006.

A. Guha, J. Matthews, R. B. Findler, and S. Krishnamurthi. Relationally-Parametric Polymorphic Contracts. In *Proceedings of the 2007 symposium on Dynamic languages - DLS '07*, 2007.

S. P. Harbison and G. L. J. Steele. *C: A Reference Manual*. 1984.

D. Herman. *A Theory of Typed Hygienic Macros*. PhD thesis, Northeastern University, Jan. 2010.

D. Herman. *Effective JavaScript*. 68 Specific Ways to Harness the Power of JavaScript. Addison-Wesley, Nov. 2012.

D. Herman and M. Wand. A Theory of Hygienic Macros. In *ESOP'08/ETAPS'08: Proceedings of the Theory and practice of software, 17th European conference on Programming languages and systems*, 2008.

R. Hinze, J. Jeuring, and A. Löh. Typed Contracts for Functional Programming. In *Functional and Logic Programming*. 2006.

E. C. M. A. International. *ECMA-262 ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), 5.1 edition, 2011. URL `http://www.ecma-international.org/publications/standards/Ecma-262.htm`.

E. C. M. A. International. *ECMA-262 ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems), 6 edition, 2015. URL `http://www.ecma-international.org/publications/standards/Ecma-262.htm`.

M. Karaorman, U. Hölzle, and J. Bruno. jContractor: A Reflective Java Library to Support Design by Contract. In *Meta-Level Architectures and Reflection*, 1999.

A. Kennedy. Relational Parametricity and Units of Measure. In *Principles of Programming Languages*, 1997.

G. Kiczales. Aspect-Oriented Programming. *ACM Computing Surveys*, 1996. doi: http://doi.acm.org/10.1145/242224.242420.

G. Kiczales, J. D. Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, July 1991. ISBN 0262610744.

E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic Macro Expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming, LFP '86*, 1986.

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*, 1987.

L. Lamport. *Latex: A Document Preparation System*. Addison-wesley, 1994.

B. Lee, R. Grimm, M. Hirzel, and K. S. McKinley. Marco: Safe, Expressive Macros for Any Language. In *ECOOP'12: Proceedings of the 26th European conference on Object-Oriented Programming*, 2012.

J. Long. The ES6 Macros Project. URL `https://github.com/jlongster/es6-macros`.

N. D. Matsakis and F. S. Klock, II. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, 2014. doi: 10.1145/2663171.2663188. URL `http://doi.acm.org/10.1145/2663171.2663188`.

Matthew Flatt and PLT. Reference: Racket. Technical report, PLT Design Inc., 2010. `http://racket-lang.org/tr1/`.

K. McFarlane. Design by Contract Framework. *The Code Project*, 2002.

W. McKeeman. Resolving Typedefs in a Multipass C Compiler. *The Journal of C Language Translation*, 1991.

B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

L. A. Meyerovich, A. P. Felt, and M. S. Miller. Object Views: Fine-Grained Sharing in Browsers. In *Proceedings of the WWW 2010, Raleigh NC, USA*, 2010.

M. S. Miller and T. V. Cutsem. Catch-All Proxies. `http://wiki.ecmascript.org/doku.php?id=harmony:proxies`.

N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. *Compiler Construction*, 2003.

Paul Hudak and Simon Peyton-Jones and Philip Wadler (eds.). Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language Version 1.2. *SIGPLAN Notices*, 1992.

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

K. M. Pitman. Special Forms in Lisp. In *Proceedings of the 1980 ACM conference on LISP and functional programming - LFP '80*, 1980.

J. Rafkind. *Syntactic Extension for Languages with Implicitly Delimited and Infix Syntax.* PhD thesis, University of Utah, 2013.

J. Rafkind and M. Flatt. Honu: Syntactic Extension for Algebraic Notation Through Enforestation. In *Generative Programming and Component Engineering, GPCE'12*, 2012.

J. Riehl. Language Embedding and Optimization in Mython. *DLS*, 2009.

D. S. Rosenblum. A Practical Approach to Programming with Assertions. *IEEE Transactions on Software Engineering*, 1995.

T. Sheard and S. L. P. Jones. Template Meta-Programming for Haskell. *SIGPLAN Notices*, 2002.

T. Sheard, Z. Benaissa, and M. Martel. *Introduction to Multistage Programming using MetaML.* Revision, 2000.

J. Siek and W. Taha. Gradual Typing for Objects. In *ECOOP 2007–Object-Oriented Programming*, 2007.

J. G. Siek and P. Wadler. Threesomes, With and Without Blame. In *POPL*, 2010.

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

K. D. Smith, J. J. Jewett, S. Montanaro, and A. Baxter. PEP 0318 – Decorators for Functions and Methods. URL `https://www.python.org/dev/peps/pep-0318/`.

G. L. Steele. Growing a Language. *Higher-Order and Symbolic Computation*, 1999.

A. G. Stephenson, L. S. LaPiana, D. R. Mulville, P. J. Rutledge, F. H. Bauer, D. Folta, G. A. Dukeman, R. Sackheim, and P. Norvig. Mars climate orbiter mishap investigation board phase 1 report. Technical report, NASA, 1999.

B. Stroustrup. Generalizing Overloading for C++2000. Technical report, AT&T Labs, 1998.

W. Taha and T. Sheard. Multi-Stage Programming with Explicit Annotations. *PEPM*, 1997.

S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as Libraries. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, 2011.

T. Tuglular, C. A. Muftuoglu, F. Belli, and M. Linschulte. Event-Based Input Validation Using Design-by-Contract Patterns. In *20th International Symposium on Software Reliability Engineering, ISSRE'09.*, 2009.

E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9. *Domain-Specific Program Generation*, 2003.

K. Wakita, K. Homizu, and A. Sasaki. Hygienic Macro System for JavaScript and Its Light-weight Implementation Framework. In *ILC '14: Proceedings of ILC 2014 on 8th International Lisp Conference*, 2014.

A. Warth and I. Piumarta. OMeta: An Object-Oriented Language for Pattern Matching. In *DLS '07: Proceedings of the 2007 symposium on Dynamic languages*, 2007.

D. Weise and R. Crew. Programmable Syntax Macros. In *ACM SIGPLAN Notices*, 1993.

# Appendix A

# Read Proof

**Lemma 1** (Parse Equivalence on Nonterminals)**.**

*For all nonterminals in the grammar $N$, $e \in AST$, $L \in Lexeme$, $P \in T$, $b \in Bool$. Assume that:*

- $P \in \mathit{RegexPrefix}_b$
- *$P$ does end with the* `.` *token*
- *if $N$ is Program, StmtList, or SourceElements then $P = \epsilon$ and $b = \mathit{false}$*
- *if $N$ is an expression nonterminal (one of AssignExpr, BinaryExpr, LHSExpr, MemberExpr, CallExpr, PrimaryExpr) then if $L$ starts with $\texttt{\{}^l$ or $\texttt{function}^l$ then $\texttt{isExprPrefix}(P, b, l) = \mathit{true}$*

*Then:*

$$L = N_e \Rightarrow \texttt{read}(L,\ P,\ b) = N'_e$$

*Proof.* By induction on the derivation of $N_e$.

- A number of cases are straightforward and hold by induction on a subderivation. These include:
  - Rule $\mathit{Program}_e ::= \mathit{SourceElements}_e$
  - Rule $\mathit{SourceElements}_e ::= \mathit{SourceElement}_e$
  - Rule $\mathit{SourceElements}_e ::= \mathit{Stmt}_e$

  - Rule $\mathit{SourceElement}_e ::= \mathit{Function}_e$
  - Rule $\mathit{StmtList}_e ::= \mathit{Stmt}_e$
  - Rule $\mathit{AssignExpr}_e ::= \mathit{BinaryExpr}_e$
  - Rule $\mathit{BinaryExpr}_e ::= \mathit{LHSExpr}_e$
  - Rule $\mathit{LHSExpr}_e ::= \mathit{MemberExpr}_e$
  - Rule $\mathit{LHSExpr}_e ::= \mathit{CallExpr}_e$
  - Rule $\mathit{MemberExpr}_e ::= \mathit{PrimaryExpr}_e$
  - Rule $\mathit{MemberExpr}_e ::= \mathit{Function}_e$

- Case for rule $\mathit{PrimaryExpr}_{\texttt{/}r\texttt{/}} ::= \texttt{/} \cdot r \cdot \texttt{/}$

  Note that $N = \mathit{PrimaryExpr}$ and $L = \texttt{/} \cdot r \cdot \texttt{/}$.

  Since $P \in \mathit{RegexPrefix}_b$ then $\texttt{read}(\texttt{/} \cdot r \cdot \texttt{/},\ P,\ b) = \texttt{/}r\texttt{/}$ and thus $\texttt{read}(L,\ P,\ b) = \mathit{PrimaryExpr'}_{\texttt{/}r\texttt{/}}$.

- Case for rule $\mathit{BinaryExpr}_{e\ \texttt{/}\ e'} ::= \mathit{BinaryExpr}_e \cdot \texttt{/} \cdot \mathit{BinaryExpr}_{e'}$

  Note that $N = \mathit{BinaryExpr}$ and $L = \mathit{BinaryExpr}_e \cdot \texttt{/} \cdot \mathit{BinaryExpr}_{e'}$. Let:

$$\begin{array}{lcl} L' & = & \mathit{BinaryExpr}_e \\ L'' & = & \mathit{BinaryExpr}_{e'} \\ T & = & \texttt{read}(L',\ P,\ b) \end{array}$$

  Then $L = L' \cdot \texttt{/} \cdot L''$. By induction $\texttt{read}(L',\ P,\ b) = \mathit{BinaryExpr'}_e$. By Lemma 2, $T \in \mathit{DividePrefix}_b$ and so

$$\begin{array}{cl} & \texttt{read}(L' \cdot \texttt{/} \cdot L'',\ P,\ b) \\ = & T \cdot \texttt{read}(\texttt{/},\ P \cdot T,\ b) \cdot \texttt{read}(L'',\ P \cdot T \cdot \texttt{/},\ b) \\ = & T \cdot \texttt{/} \cdot \texttt{read}(L'',\ P \cdot T \cdot \texttt{/},\ b) \end{array}$$

  Since $T \cdot \texttt{/} \in \mathit{RegexPrefix}_b$ and $\texttt{isExprPrefix}(T \cdot \texttt{/}, b, l) = \mathit{true}$ for all $T$, $b$, and $l$, by induction

$$\texttt{read}(L'',\ P \cdot T \cdot \texttt{/},\ b) = \mathit{BinaryExpr'}_{e'}$$

  and thus $\texttt{read}(L,\ P,\ b) = \mathit{BinaryExpr'}_{e\ \texttt{/}\ e'}$.

- Case for rule $\textit{Stmt}_e ::= \textit{AssignExpr}_e \cdot \texttt{;}$ where lookahead $\neq$ `{` or `function`

  We have $N = \textit{Stmt}$ and $L = \textit{AssignExpr}_e \cdot \texttt{;}$. Since from the grammar production lookahead we know that $L$ does not begin with `{` or `function` this case holds by induction.

- Case for rule $\textit{SourceElements}_{e\ e'} ::= \textit{SourceElements}_e \cdot \textit{SourceElement}_{e'}$

  We have $N = \textit{SourceElements}$ and $L = \textit{SourceElements}_e \cdot \textit{SourceElement}_{e'}$. Let:

$$\begin{aligned} L' &= \textit{SourceElements}_e \\ L'' &= \textit{SourceElement}_{e'} \\ T &= \texttt{read}(L',\ P,\ b) \end{aligned}$$

  Then $L = L' \cdot L''$. By induction $T = \textit{SourceElements'}_e$. From our assumptions $P = \epsilon$ and $b = \textit{false}$ so by Lemma 3, $T \in \textit{RegexPrefix}_b$ and $P \cdot T \in \textit{RegexPrefix}_b$ (since by definition $\epsilon \cdot T$ is still in $\textit{RegexPrefix}_b$). By Lemma 5, $T$ does not end with a `.` token. Thus by induction $\texttt{read}(L'',\ P \cdot T,\ b) = \textit{SourceElement'}_{e'}$ and thus $\texttt{read}(L,\ P,\ b) = \textit{SourceElements'}_{e\ e'}$

- Case for rule $\textit{Stmt}_{x\texttt{:}e} ::= x \cdot \texttt{:} \cdot \textit{Stmt}_e$

  Note that $N = \textit{Stmt}$ and $L = x \cdot \texttt{:} \cdot \textit{Stmt}_e$. Let:

$$\begin{aligned} L' &= \textit{Stmt}_e \\ T &= \texttt{read}(L',\ P \cdot x \cdot \texttt{:},\ b) \end{aligned}$$

  Then $L = x \cdot \texttt{:} \cdot L'$. Since $P \cdot x \cdot \texttt{:} \in \textit{RegexPrefix}_b$ by induction $T = \textit{Stmt'}_e$ and thus $\texttt{read}(L,\ P,\ b) = \textit{Stmt'}_{x\texttt{:}e}$.

- Case for rule $\textit{Stmt}_{\texttt{if}\ \texttt{(}e\texttt{)}\ e'} ::= \texttt{if} \cdot \texttt{(} \cdot \textit{AssignExpr}_e \cdot \texttt{)} \cdot \textit{Stmt}_{e'}$

  Note that $N = \textit{Stmt}$ and $L = \texttt{if} \cdot \texttt{(} \cdot \textit{AssignExpr}_e \cdot \texttt{)} \cdot \textit{Stmt}_{e'}$. Let:

$$\begin{aligned} L' &= \textit{AssignExpr}_e \\ L'' &= \textit{Stmt}_{e'} \\ T &= \texttt{read}(L',\ \epsilon,\ \textit{true}) \\ T' &= \texttt{read}(L'',\ P \cdot \texttt{if} \cdot \underline{(T)},\ b) \end{aligned}$$

  Then $L = \texttt{if} \cdot \texttt{(} \cdot L' \cdot \texttt{)} \cdot L''$. Since from its definition $\texttt{isExprPrefix}(\epsilon,\ \textit{true},\ l) = \textit{true}$ for all $l$, by induction $T = \textit{AssignExpr'}_e$. From our assumptions $P$ does not end with the `.` token, so $P \cdot \texttt{if} \cdot \underline{(T)} \in \mathit{RegexPrefix}_b$ thus by induction $T' = \mathit{Stmt'}_{e'}$. Therefore $\texttt{read}(L,\ P,\ b) = \mathit{Stmt'}_{\texttt{if (}e\texttt{)}\ e'}$.

- Case for rule $\mathit{Stmt}_{\texttt{return}} ::= \texttt{return}$

  Note that $N = \mathit{Stmt}$ and $L = \texttt{return}$. Follows directly.

- Case for rule $\mathit{Stmt}_{\texttt{return}\ e} ::= \texttt{return} \cdot$ [no line terminator here] $\mathit{AssignExpr}_e \cdot \texttt{;}$

  Note that $N = \mathit{Stmt}$ and $L = \texttt{return}^l \cdot \mathit{AssignExpr}_e \cdot \texttt{;}$. Let:

$$\begin{array}{rcl} L' & = & \mathit{AssignExpr}_e \\ T & = & \texttt{read}(L',\ P \cdot \texttt{return}^l,\ b) \end{array}$$

  Then $L = \texttt{return} \cdot L' \cdot \texttt{;}$. Since from this production rule there is no line break after the `return` keyword, $\texttt{isExprPrefix}(P \cdot \texttt{return}^l, b, l) = \mathit{true}$ and $P \cdot \texttt{return} \in \mathit{RegexPrefix}_b$ and and so by induction, $T = \mathit{AssignExpr'}_e$. Thus $\texttt{read}(L,\ P,\ b) = \mathit{Stmt'}_{\texttt{return}\ e}$.

- Case for rule $\mathit{StmtList}_{e\ e'} ::= \mathit{StmtList}_e \cdot \mathit{Stmt}_{e'}$

  We have $N = \mathit{StmtList}$ and $L = \mathit{StmtList}_e \cdot \mathit{Stmt}_{e'}$. Let:

$$\begin{array}{rcl} L' & = & \mathit{StmtList}_e \\ L'' & = & \mathit{Stmt}_{e'} \\ T & = & \texttt{read}(L',\ P,\ b) \end{array}$$

  Then $L = L' \cdot L''$. By induction $T = \mathit{StmtList'}_e$ and by Lemma 4, $T \in \mathit{RegexPrefix}_b$. From our assumptions $P = \epsilon$ so $P \cdot T \in \mathit{RegexPrefix}_b$ (since by definition $\epsilon \cdot T$ is still in $\mathit{RegexPrefix}_b$). From Lemma 5, $T$ does not end with a `.` token. Thus by induction $\texttt{read}(L'',\ P \cdot T,\ b) = \mathit{Stmt'}_{e'}$ and thus $\texttt{read}(L,\ P,\ b) = \mathit{StmtList'}_{e\ e'}$.

- Case for rule $\mathit{Stmt}_{\{e\}} ::= \texttt{\{} \cdot \mathit{StmtList}_e \cdot \texttt{\}}$

  Note that $N = \mathit{Stmt}$ and $L = \texttt{\{} \cdot \mathit{StmtList}_e \cdot \texttt{\}}$. Let:

$$\begin{array}{rcl} L' & = & \mathit{StmtList}_e \\ T & = & \texttt{read}(L',\ \epsilon,\ \mathit{false}) \end{array}$$

  Then $L = \texttt{\{} \cdot L' \cdot \texttt{\}}$. By induction $T = \mathit{StmtList'}_e$ and thus $\texttt{read}(L,\ P,\ b) = \mathit{Stmt'}_{\{e\}}$.

- Case for rule $\textit{Function}_{\texttt{function}\ x\ \texttt{(}x'\texttt{)}\ \texttt{\{}e\texttt{\}}} ::= \texttt{function}\cdot x\cdot \texttt{(}\cdot x'\cdot \texttt{)}\cdot \texttt{\{}\cdot \textit{SourceElements}_e\cdot \texttt{\}}$

  Note that $N = \textit{Function}$ and $L = \texttt{function} \cdot x \cdot \texttt{(} \cdot x' \cdot \texttt{)} \cdot \texttt{\{} \cdot \textit{SourceElements}_e \cdot \texttt{\}}$. Let:

$$\begin{array}{rcl} L' & = & \textit{SourceElements}_e \\ T & = & \texttt{read}(L',\ \epsilon,\ \textit{false}) \end{array}$$

  Then $L = \texttt{function} \cdot x \cdot \texttt{(} \cdot x' \cdot \texttt{)} \cdot \texttt{\{} \cdot L' \cdot \texttt{\}}$. By induction $T = \textit{SourceElements'}_e$. Thus $\texttt{read}(L,\ P,\ b) = \textit{Function'}_{\texttt{function}\ x\ \texttt{(}x'\texttt{)}\ \texttt{\{}e\texttt{\}}}$.

- Case for rule $\textit{PrimaryExpr}_{\texttt{\{}x\texttt{:}e\texttt{\}}} ::= \texttt{\{} \cdot x \cdot \texttt{:} \cdot \textit{AssignExpr}_e \cdot \texttt{\}}$

  Note that $N = \textit{PrimaryExpr}$ and $L = \texttt{\{}^l \cdot x \cdot \texttt{:} \cdot \textit{AssignExpr}_e \cdot \texttt{\}}$ Let: $L' = \textit{AssignExpr}_e$. Then $L = \texttt{\{}^l \cdot x \cdot \texttt{:} \cdot L' \cdot \texttt{\}}$. Since from our assumptions $\texttt{isExprPrefix}(P, b, l) = \textit{true}$ then $T = \texttt{read}(L',\ x \cdot \texttt{:},\ \textit{true})$ and $x \cdot \texttt{:} \in \textit{RegexPrefix}_{\textit{true}}$ and $\texttt{isExprPrefix}(x \cdot \texttt{:}, \textit{true}, l')$ for all $l'$.

  So by induction $T = \textit{AssignExpr'}_e$ and thus $\texttt{read}(L,\ P,\ b) = \textit{PrimaryExpr'}_{\texttt{\{}x\texttt{:}e\texttt{\}}}$.

- Case for rule $\textit{PrimaryExpr}_{\texttt{(}e\texttt{)}} ::= \texttt{(} \cdot \textit{AssignExpr}_e \cdot \texttt{)}$

  Note that $N = \textit{PrimaryExpr}$ and $L = \texttt{(} \cdot \textit{AssignExpr}_e \cdot \texttt{)}$. Let:

$$\begin{array}{rcl} L' & = & \textit{AssignExpr}_e \\ T & = & \texttt{read}(L',\ \epsilon,\ \textit{true}) \end{array}$$

  Then $L = \texttt{(} \cdot L' \cdot \texttt{)}$. Since from its definition $\texttt{isExprPrefix}(\epsilon, \textit{true}, l) = \textit{true}$ for all $l$, by induction $T = \textit{AssignExpr'}_e$. Thus, $\texttt{read}(L,\ P,\ b) = \textit{PrimaryExpr'}_{\texttt{(}e\texttt{)}}$.

- Case for rule $\textit{PrimaryExpr}_x ::= x$

  Note that $N = \textit{PrimaryExpr}$ and $L = x$. Follows directly.

- Case for rule $\textit{CallExpr}_{e\texttt{(}e'\texttt{)}} ::= \textit{CallExpr}_e \cdot \texttt{(} \cdot \textit{AssignExpr}_{e'} \cdot \texttt{)}$

  Note that $N = \textit{CallExpr}$ and $L = \textit{CallExpr}_e \cdot \texttt{(} \cdot \textit{AssignExpr}_{e'} \cdot \texttt{)}$. Let:

$$\begin{array}{rcl} L' & = & \textit{CallExpr}_e \\ L'' & = & \textit{AssignExpr}_{e'} \\ T & = & \texttt{read}(L',\ P,\ b) \\ T' & = & \texttt{read}(L'',\ \epsilon,\ \textit{true}) \end{array}$$

Then $L = L' \cdot \texttt{(} \cdot L'' \cdot \texttt{)}$. By induction $T = \mathit{CallExpr'}_e$ and since from its definition $\texttt{isExprPrefix}(\epsilon,\ \mathit{true},\ l) = \mathit{true}$ for any $l$, by induction $T' = \mathit{AssignExpr'}_{e'}$. Thus, $\texttt{read}(L,\ P,\ b) = \mathit{CallExpr}_{e\texttt{(}e'\texttt{)}}$.

- Case for rule $\mathit{CallExpr}_{e\texttt{.x}} ::= \mathit{CallExpr}_e \cdot \texttt{.} \cdot \texttt{x}$

  Note that $N = \mathit{CallExpr}$ and $L = \mathit{CallExpr}_e \cdot \texttt{.} \cdot \texttt{x}$. Holds by induction on $\mathit{CallExpr}$.

- Case for rule $\mathit{MemberExpr}_{e\texttt{.x}} ::= \mathit{MemberExpr}_e \cdot \texttt{.} \cdot \texttt{x}$

  Note that $N = \mathit{MemberExpr}$ and $L = \mathit{MemberExpr}_e \cdot \texttt{.} \cdot \texttt{x}$. Holds by induction on $\mathit{MemberExpr}$.

- Case for rule $\mathit{AssignExpr}_{e\ \texttt{=}\ e'} ::= \mathit{LHSExpr}_e \cdot \texttt{=} \cdot \mathit{AssignExpr}_{e'}$

  Note that $N = \mathit{AssignExpr}$ and $L = \mathit{LHSExpr}_e \cdot \texttt{=} \cdot \mathit{AssignExpr}_{e'}$. Let:

$$\begin{array}{rcl} L' & = & \mathit{LHSExpr}_e \\ L'' & = & \mathit{AssignExpr}_{e'} \\ T & = & \texttt{read}(L',\ P,\ b) \end{array}$$

  Then $L = L' \cdot \texttt{=} \cdot L''$. By induction, $T = \mathit{LHSExpr'}_e$. Since $T \cdot \texttt{=} \in \mathit{RegexPrefix}_b$ and $\texttt{isExprPrefix}(T \cdot \texttt{=},\ b,\ l) = \mathit{true}$ for all $T$, $b$, $l$, by induction

$$\texttt{read}(L'',\ P \cdot T \cdot \texttt{=},\ b) = \mathit{AssignExpr'}_{e'}$$

  and thus $\texttt{read}(L,\ P,\ b) = \mathit{AssignExpr'}_{e\ \texttt{=}\ e'}$

- Case for rule $\mathit{BinaryExpr}_{e\ \texttt{+}\ e'} ::= \mathit{BinaryExpr}_e \cdot \texttt{+} \cdot \mathit{BinaryExpr}_{e'}$

  Note that $N = \mathit{BinaryExpr}$ and $L = \mathit{BinaryExpr}_e \cdot \texttt{+} \cdot \mathit{BinaryExpr}_{e'}$. Let:

$$\begin{array}{rcl} L' & = & \mathit{BinaryExpr}_e \\ L'' & = & \mathit{BinaryExpr}_{e'} \\ T & = & \texttt{read}(L',\ P,\ b) \end{array}$$

  Then $L = L' \cdot \texttt{+} \cdot L''$. By induction $\texttt{read}(L',\ P,\ b) = \mathit{BinaryExpr'}_e$. Since $T \cdot \texttt{+} \in \mathit{RegexPrefix}_b$ and $\texttt{isExprPrefix}(T \cdot \texttt{+},\ b,\ l) = \mathit{true}$ for all $T$, $b$, $l$, by induction

$$\texttt{read}(L'',\ P \cdot T \cdot \texttt{+},\ b) = \mathit{BinaryExpr'}_{e'}$$

  and thus $\texttt{read}(L,\ P,\ b) = \mathit{BinaryExpr'}_{e\ \texttt{+}\ e'}$.

- Case for rule $\textit{CallExpr}_{e\texttt{(}e'\texttt{)}} ::= \textit{MemberExpr}_e \cdot \texttt{(} \cdot \textit{AssignExpr}_{e'} \cdot \texttt{)}$

  Note that $N = \textit{CallExpr}$ and $L = \textit{MemberExpr}_e \cdot \texttt{(} \cdot \textit{AssignExpr}_{e'} \cdot \texttt{)}$. Let:

$$\begin{aligned} L' &= \textit{MemberExpr}_e \\ L' &= \textit{AssignExpr}_{e'} \\ T &= \texttt{read}(L',\ P,\ b) \\ T' &= \texttt{read}(L'',\ \epsilon,\ \textit{true}) \end{aligned}$$

  Then $L = L' \cdot \texttt{(} \cdot L'' \cdot \texttt{)}$. By induction $T = \textit{MemberExpr'}_e$ and since from its definition $\texttt{isExprPrefix}(\epsilon,\ \textit{true},\ l) = \textit{true}$ for all $l$, by induction $T' = \textit{AssignExpr'}_{e'}$. Thus, $\texttt{read}(L,\ P,\ b) = \textit{CallExpr}_{e\texttt{(}e'\texttt{)}}$.

□

**Lemma 2** (Binary Divide Prefix)**.**

*For all $L \in \textit{Lexeme}$, $P \in \textit{Token}$, $b \in \textit{Bool}$, $e \in \textit{AST}$. Assuming that:*

- *$P \in \textit{RegexPrefix}_b$, and*
- *if $L$ starts with either $\texttt{\{}^l$ or $\texttt{function}^l$ then $\texttt{isExprPrefix}(P,\ b,\ l) = \textit{true}$*

*Then:*

$$\texttt{read}(L,\ P,\ b) = \textit{BinaryExpr'}_e \Rightarrow \texttt{read}(L,\ P,\ b) \in \textit{DividePrefix}_b.$$

*Proof.* By induction on the derivation of $\textit{BinaryExpr'}_e$. A few example cases are shown. The others are similar.

- Case for rule $\textit{PrimaryExpr'}_x ::= x$

  Note that $\texttt{read}(L,\ P,\ b) = x$. Holds since $P \cdot x \in \textit{DividePrefix}_b$.

- Case for rule $\textit{PrimaryExpr'}_{\texttt{/}x\texttt{/}} ::= \texttt{/}x\texttt{/}$

  Note that $\texttt{read}(L,\ P,\ b) = \texttt{/}x\texttt{/}$.

  Holds since $P \cdot \texttt{/}x\texttt{/} \in \textit{DividePrefix}_b$.

- Case for rule $\textit{PrimaryExpr'}_{\texttt{\{}x\texttt{:}e\texttt{\}}} ::= \texttt{\{}^l \cdot x \cdot \texttt{:} \cdot T \cdot \texttt{\}}$

  Note that $\texttt{read}(L,\ P,\ b) = \texttt{\{}^l \cdot x \cdot \texttt{:} \cdot T \cdot \texttt{\}}$.

  Since $\texttt{isExprPrefix}(P,\ b,\ l) = \textit{true}$ we have $P \cdot \underline{\texttt{\{}x \cdot \texttt{:} \cdot T\texttt{\}}} \in \textit{DividePrefix}_b$.

□

**Lemma 3** (SourceElements Regexp Prefix)**.**

*For all* $e \in AST$.

$\mathit{SourceElements'}_e \in \mathit{RegexPrefix}_{\mathit{false}}$.

*Proof.* By induction on the derivation of $\mathit{SourceElements'}_e$. □

**Lemma 4** (StmtList Regexp Prefix)**.**

*For all* $e \in AST$.

$\mathit{StmtList'}_e \in \mathit{RegexPrefix}_{\mathit{false}}$.

*Proof.* By induction on the derivation of $\mathit{StmtList'}_e$. □

**Lemma 5** (Nonterminals do not end with dot)**.**

*For all nonterminals in the grammar* $N'$, $L \in \mathit{Lexeme}$, $P \in \mathit{Token}$, $b \in \mathit{Bool}$, $e \in AST$.

*If* $\texttt{read}(L,\ P,\ b) = N'_e$ *then* $\texttt{read}(L,\ P,\ b)$ *does not end with the* `.` *token.*

*Proof.* By induction on the derivation of $N'$. Note that the only productions in the grammar with `.` are *MemberExpr'* and *CallExpr'* and in both cases `.` does not end the production. □

**Lemma 6** (`read` is well-founded)**.** *`read` is a well-founded function (at most one case applies to a given input).*

*Proof.* By case analysis and the fact that $\mathit{RegexPrefix}_b$ and $\mathit{DividePrefix}_b$ are disjoint. □