Hygienic Macro System for JavaScript and Its Light-weight Implementation Framework

Ken Wakita
Tokyo Institute of Technology
wakita@is.titech.ac.jp

Kanako Homizu
Independent Researcher
homizu.k@gmail.com

Akira Sasaki
Hosei University
asasaki@hosei.ac.jp

ABSTRACT

In spite of its soundness, ease of use, and descriptive power, few hygienic macro systems have been incorporated to non-LISP programming languages. The difficulty lies in the self-describing nature of the macro system. Proposed is the design and a systematic implementation framework of hygienic macro systems for general programming languages. In addition to the standard syntax-rules macro features, our macro system incorporates handling of punctuations and keywords, introduces invisible blocks of code called phantom groups, and proposes the notion of suffix pattern to address problems due to recursive macro definitions. These features have been introduced to deal with richer syntax of general programming languages. The proposed implementation framework incorporates staged parser architecture, a parser generation technique using parsing expression grammars, and mutual conversion technique between the macro-enabled host language and Scheme that allows our macro system to delegate actual macro expansion task to a Scheme interpreter, instead of building hygienic macro expander from scratch. ExJS, our prototype implementation of a hygienic macro system for JavaScript, is implemented in less than 2,000 lines of JavaScript and Scheme, and exhibits its flexible syntactic extensibility.

Categories and Subject Descriptors

D.3.4 [Programming Languages]: Processors—*Preprocessors*

General Terms

Programming languages

Keywords

Hygienic Macro Systems, Parser, Program Transformation, Parsing Expression Grammar, JavaScript, Scheme

1. INTRODUCTION

A *macro system* [18] is a compile-time meta programming tool for extending the language syntax. A *macro* is a syntactic form

introduced to the host programming language via a programmer-defined *macro definition*. An occurrence of such macro definition is called a *macro form* or a *macro usage*. Using the macro forms, a programmer can enjoy a higher-level, abstract, declarative, and robust style of programming.

A macro definition also creates a function called *macro expander*, or *macro transformer*, that transforms a macro form written in the extended syntax to the one described in the standard syntax of the host programming language. Hereafter, we will call the standard programming language with no syntactic extensibility the *host language* and the one that offers macro capability the *macro-enabled language* (of the host language).

Unlike traditional *lexical* or *token-level* macro systems such as Unix `m4` [22], C preprocessor (CPP) [15], and $\text{\TeX}$ [16], *syntactic macro systems* (e.g., Common Lisp [26], programmable syntax macros [27], Template Haskell [24], `camlp5` [9]) respect the syntax of the host language and solve the problem of lexical macro systems. Most of the syntactic macro systems offer lower-level programming interface to manipulate the abstract syntax tree. An advantage with homoiconic languages like LISP is that their program structure is naturally expressed in terms of their native data representation called *S-expression* and therefore the LISP readily offers access to their abstract syntax trees. On the other hand, with languages that have more conventional syntax, the abstract syntax trees are complex and are only accessible via complicated API.

The *hygienic macro system* [17, 4, 5, 10], proposed for Scheme [6], pays close attention to sound treatment of variable bindings and carefully avoids accidental name collisions between variables used inside a macro definition and those that are bound in the lexical environment. Hygienic macro system freed the programmer from error-prone name management and potential danger of name clash.

Another advantage of hygienic macro systems is their easy-to-use, *template-based style* of macro programming. In hygienic macro systems, a macro is defined by a pair of a *pattern* and a *template*, where the pattern defines the syntax of the macro and the template illustrates the replacement for the matching macro usage. In this way, the macro programmer can just declare the syntax of the macro form, and are freed from implementing the parser and the transformer for the macro. The hygienic macro expander is responsible for the construction of the parser and macro transformer function for the macro.

In spite of their soundness and ease of programming, hygienic macro systems have not been incorporated into most programming languages; we see them only in various Scheme implementations and a few research prototypes of other languages.

There are technical obstacles for introduction of hygienic macro systems to general programming languages that offer richer and more stylish syntax. The first is related to parsing. Imagine a

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ILC '14 August 14 - 17 2014, Montreal, QC, Canada

Copyright 2014 ACM 978-1-4503-2931-6/14/08 ...\$15.00

<http://dx.doi.org/10.1145/2635648.2635653>.

macro-enabled program that contains a macro definition and its usages. Because the program contains macro forms, the parser that processes them must recognize the macro syntax. To make things more complex, the syntax of the macro form is defined via the macro definition. At this point, we are put in a chicken-and-egg situation. This is not a problem for LISP because it imposes parenthesized meta-syntax to its macro-extended forms. That means the standard LISP parser can parse the macro-enabled LISP code. Previous work on hygienic macro systems for non-LISP languages also imposed strict syntactic restriction over the meta forms via meta-syntax, and allowed the programmer to write macro forms whose syntax resembles the function applications or existing control structures. Such syntactic restriction disallows us from using a fancy macro form like “`let x = e1 and y = e2 in x*x+y*y`” and forces us to endure a disgraceful alternative like “`let((x = e1), (y = e2)), (x*x+y*y))`”. In order to let the programmer add fancy macro forms, the implementation of extensible syntactic macro for a general programming language requires flexible parser extensibility as well as a solution to the self-describing nature of the macro-enabled programs.

The paper proposes a hygienic macro system for a general programming language and its systematic implementation framework. The design of the macro system follows that of the R⁶RS-style macro system. Additionally, to deal with richer syntax of a general programming language, the proposed macro system offers *syntax class annotations*, *phantom grouping* in repetitive forms, *suffix patterns* to address problems due to recursively defined macros, and flexible treatment of punctuations and literal keywords.

In the implementation, the self-describing nature of the macro-enhanced host languages has been addressed by a *staged parsing* method. The 1st-stage parser collects macro definitions, recognizes the syntax of macro forms, and generates the 2nd-stage parser. This 2nd-stage parser fully recognizes the syntax of the macro-enabled program and processes it. We have chosen parsing expression grammar [14] to describe the syntax of the host and used PEG.js as a tool to generate the macro-aware parser from the syntax descriptions.

In our framework, macro expansion is delegated to the macro expander of a Scheme interpreter, following the technique that we have previously proposed [2]. The macro-enhanced program described in the host programming language is converted to S-expression, which is expanded by the macro expander of Scheme to generate a macro-free S-expression, which finally is converted back to the host programming language.

We have applied this framework to implement a hygienic macro system for JavaScript. The implemented macro system is an R⁶RS-style (namely, *syntax-rules*), template-based, hygienic macro system. Our simple framework enabled us to build the macro system with a compact code base of 2,000 lines of JavaScript and Scheme.

The proposed implementation framework decouples the implementation of the macro expander from the implementation of the macro system, and delegates the task of macro expansion to R⁶RS Scheme. The implementation framework, combined with the PEG-based implementation technique for the extensible parser, results in a simple, elegant, and systematic implementation strategy of hygienic macro systems for general programming languages.

The rest of the paper is structured as follows: Section 2 illustrates the core features of the ExJS macro system, Section 3 describes the implementation scheme of ExJS, Section 4 presents a few additional features of ExJS, Section 5 is for discussion, Section 6 compares our proposal with related work, and the last section summarizes this paper.

```

1 statement unless {
2   expression: C;
3   statement: S;
4   { unless (C) S => if (!C) S }
5 }
6
7 var c1 = true;
8 function f(c1, c2) {
9   if (c1)
10     unless (c2)
11       console.log('P1');
12   else console.log('P2');
13 }
14 f(true, true); // No output
15 f(true, false); // P1

```

(a)

```

1 var c1 = true;
2 function f(c1_199_, c2_199_) {
3   if (c1_199_) {
4     if (!! c2_199_)
5       ((console.log) ("P1"));
6   } else ((console.log) ("P2"));
7 }
8 f(true, true);
9 f(true, false);

```

(b)

Figure 1: Macro-enhanced JavaScript code (top) is expanded into a plain JavaScript code (bottom)

2. EXJS MACRO SYSTEM

This section introduces our hygienic macro system proposed for JavaScript, named *ExJS*. The design of ExJS is strongly influenced by the *syntax-rules* macro system of R⁶RS. A macro writer can define a new macro according to a source-level, template-driven pattern language. The syntax of ExJS macro system is simple but in comparison with *syntax-rules*, it is more involved to deal with JavaScript’s richer syntax. The syntax of JavaScript consists of expressions and statements. ExJS offers two kinds of declaration syntax, namely the **expression** class and the **statement** class, to define a new expression and statement syntax, respectively.

Figure 1-(a) presents a definition of **unless** statement macro that introduces a statement that expresses negative, and conditional execution. We can read the definition as the following:

*Let C and S be syntactic meta-variables that denote expressions and statements, respectively. The syntax of the **unless** macro form is defined by the left hand side of =>, and is transformed to an **if** statement as given to the right of =>.*

We call the left hand side of => the *pattern* part and its right hand side the *replacement template* part, or simply the *template* part.

The attribution of syntactic classes to meta symbols, namely **expression** class and **statement** class, are used by ExJS to generate respective parsers that recognize respective subforms. For example, ExJS calls the expression parser for parsing the part denoted by C and the statement parser for S.

Such defined new syntactic form is used in line 10. A usage of a statement macro belongs to the statement class of the JavaScript

```

1 expression let {
2   identifier: v;
3   expression: e, body;
4   keyword: In;
5   { let (v = e), ... In body
6     => (function (v, ...) {
7       return body; })(e, ...) }
8 }
9 var where = 'global', a = 1, b = 2;
10 let (where = 'local'), (a = 100) In
11   console.log(where, a, b); // local 100 2
12 console.log(where, a, b);    // global 1 2

```

Figure 2: A definition of **let** macro and its usage.

grammar. The meta-variables in the macro template pattern can be regarded as placeholders for the subforms contained in the macro usage. Expansion of a macro usage is replacement of the macro usage with the template part of the macro definition, while occurrences of meta-variables in the template are substituted by the matching subforms within the macro usage.

Figure 1-(b) shows how ExJS expands this macro-enabled JavaScript code to plain JavaScript code. Because ExJS is a syntactic macro system, its macro expander preserves the syntactic structures of expressions and statements contained in macro forms. For example, the use of **unless** macro in Figure 1-(a):line 10 occurs inside an **if** statement. Its textual macro expansion would introduce a new **if** statement that accidentally capture the **else** clause of the (original) **if** statement. In contrast, ExJS performs macro expansion within the **then** clause of the (original) **if** statement properly and preserves the syntactic structure of the original code (see Figure 1-(b): lines 3-6).

Being a hygienic macro expander, ExJS renames bound variables (e.g., from *c1* to *c1_199_*) to avoid accidental name capturing of variables by macro-introduced variables.

2.1 Example (let)

Expression macros are defined in a similar fashion as statement macros. Figure 2 defines the **let** macro that makes local variable bindings. The syntactic form of this macro is given in line 5 and its replacement template in lines 6-7. The macro establishes local bindings by an application of anonymous function. In this example, two new syntactic classes, namely **identifier** and **keyword**, are used. An **identifier** meta-variable (e.g., *v*) stands for JavaScript identifiers and a **keyword** meta-variable (e.g., **In**) for a user-defined literal keywords that are used within this macro definition.

2.1.1 Repetition

Sometimes we would like to define a syntax that allows repetition. ExJS inherits the dots (*...*) notation from R⁶RS to denote zero or more repetition of the preceding form. A repetition pattern is either a meta variables or a compound pattern that is optionally followed by three dots. In the latter case, the macro writer surrounds the form with the one of three pairs of grouping symbols, namely ' (' and ') ', { and }, or [and], or a pair of *phantom grouping* symbols, namely [# and #].

For example, the variable binding part of the **let** macro comprises a repetition of a compound pattern that stands for a variable initialization (e.g., “(*v* = *e*)”) followed by dots. This pattern definition introduces a new syntactic form as exemplified by the macro usage in Figure 2, line 10.

In contrast to Scheme whose repetition syntax is a list enclosed

```

1 expression Append {
2   expression: DOM, c1, c2;
3   keyword: to;
4   { Append c1 to DOM => DOM.append(c1) }
5   { Append c1, c2, ... to DOM
6     => Append c2, ... to DOM.append(c1) }
7 }
8 var pageBody =
9   Append Title, Menu, Content to $('body');

```

Figure 3: A definition of **Append** macro and its usage.

by a pair of parentheses, a sequence of expressions in JavaScript can be separated by punctuation symbols like commas and semi-colons. ExJS allows the macro writer to choose arbitrary sequence of tokens to separate sequences of forms. ExJS treats the sequence of tokens between the dots and its preceding subpattern as punctuation literal symbols. Such punctuation symbol can be just a symbol, a multi-letter symbol, and an alphabetical symbol (possibly in UTF8 encoding). The variable binding form of **let** macro is interpreted as a comma-separated zero or more sequence of assignments which are surrounded by parentheses.

2.1.2 Phantom Groups

Unlike other grouping symbols, phantom grouping symbols are virtual entity and are used only in the repetition patterns, and they do not occur in macro usages. The purpose of phantom groups is to specify the pattern that is to be repeated in a repetition pattern.

Suppose for example that we would like to remove the awkward parentheses from **let** usage forms. You might be tempted to replace the macro form syntax with the following:

```
{ let v = e, ... In body }
```

However this definition does not work because it defines a macro form where **let**, a variable name (*v*), and an equal symbol is followed by a sequence of expressions separated with commas as follows.

```
let v = e1, e2, e3 In body
```

We, instead, would like to define a syntax where a **let** followed by a sequence of assignments separated by commas. A phantom group is used to explicitly tell ExJS that the pattern of repetition is an assignment and not a sequence of expressions as follows.

```
{ let [# v = e #], ... In body }
```

ExJS regards [# and #] as a pair of *invisible* parentheses. With this revision, we can remove awkward parentheses from the previous usage of **let** macro as follows.

```
let where = 'local', a = 100 In
console.log(where, a, b);
```

2.2 Example (Append)

2.2.1 Multiple patterns

ExJS supports a macro definition that contains multiple patterns and recursive macro usage. Figure. 3 defines the **Append** macro that takes HTML DOM elements and appends them to the HTML body. The first pattern (line 4) deals with a case when the macro form is given a single element. When the macro form is supplied with more elements, it is expanded using the second pattern (lines 5-6).

Multiple patterns that occur in the same macro definition are prioritized according to their order of appearance.

To avoid ambiguity, meta variables must occur uniquely in the pattern part of the macro definition. On the other hand, they can occur multiple times in the replacement template. In such cases, the macro can duplicate the sub-form bound to the meta-variable in the macro form.

2.2.2 Suffix patterns

Line 6 of Figure 3 contains a recursive usage of the **Append** macro. It should be noted that none of the two patterns of the **Append** macro definition exactly matches this usage (i.e., “*c1*” and “*c1, c2, ...*” versus “*c2, ...*”). ExJS allows the suffix of a repetition pattern to occur in the replacement template of the macro definitions. This relaxation allows for the following forms.

```
Append c1 to DOM
Append c1, c2, ... to DOM
Append , c2, ... to DOM
Append c2, ... to DOM
```

The first two forms have the same structure as the two patterns of the **Append** macro. The rest are suffixes of the second pattern. The fourth in the list of suffix patterns corresponds to the macro usage in line 6 of Figure 3. Introduction of the *suffix pattern* allows for recursive macros in ExJS programming. This syntactic relaxation, allows the macro programmer to write a recursive macro that contains macro subforms.

It should be noted that the extended syntax applies only to the template part of macro definitions. The macro usage must follow the syntax of macro pattern forms. More specifically, for example, the form “**Append**, c, ..., **to** DOM” is disallowed to occur outside the macro template because (1) the macro pattern does not allow an **Append** followed by a comma and (2) it includes ellipsis, which is a token only allowed in the template part.

The list of the suffix patterns can sometimes contain a funny pattern such as the third in the list where **Append** is immediately followed by a comma. Because this extended syntax is allowed only in the template part, macro users’ accidental injury from using this pattern is avoided. If the macro writer accidentally uses this form, macro expansion generates a macro form where a comma follows **Append**. Because none of the macro forms matches this pattern, macro expansion fails. Elimination of suffix patterns which are not derivable from macro forms remains a future work.

3. IMPLEMENTATION

As we have mentioned previously, the main difficulty for implementing hygienic macro system is twofold: the syntactic macro system requires the parser of the host language to be syntactically extensible, plus implementation of efficient hygienic macro system is hard. Our previous attempt approached these problems by an ad-hoc implementation of extensible parser relying on incremental extension of a top-down operator precedence parser [20, 7] and mutual transformation of macro-extended JavaScript to/from Scheme to make use of existing macro expander implementation.

In this paper, we propose a systematic method of building an extensible parser using a staged parsing architecture and a parser generator [14] for parsing expression grammars, combined with our previous mutual transformation technique.

3.1 The Outline

The proposed implementation scheme consists of two phases: the *parser extension phase* and the *macro expansion phase* (Fig-

1. Parser Extension Phase

- 1st state parser analyzes the syntax of macro forms
- Generate PEG grammars for the defined macros
- Combine JavaScript grammar and macro grammars and generate macro-aware 2nd-stage parser

2. Macro Expansion Phase

- Parse the macro-enabled program using the 2nd-stage parser
- Convert the AST to S-expression
- Macro-expand the S-expression
- Convert the S-expression back to macro-free JavaScript

Figure 4: The outline of macro processing

Figure 5: The structure of macro-enabled JavaScript program is surprisingly complex

ure 4). The macro expander takes a macro-enabled JavaScript program which contains macro definitions and macro usages as well as standard JavaScript code. The first task of this phase is the retrieval of all the macro patterns from the program by the 1st stage parser: Phase 1(a). As we have noted in the previous section, the syntax of a macro is defined by the pattern part of its definition. According to patterns contained in the program, a PEG grammar for the macro is generated: Phase 1(b). All the macro grammars are merged in with the grammar of JavaScript to make a macro-syntax-aware grammar. This extended grammar is passed to PEG-based parser generator to build a macro-aware 2nd-stage parser: Phase 1(c). In our current implementation, we use David Majda’s PEG.js¹ for this purpose.

The macro-aware 2nd-stage parser is used to re-parse the macro-enabled JavaScript program: Phase 2(a). The abstract syntax tree is converted to S-expression: Phase 2(b), and is expanded by the macro expander of Scheme to obtain a macro-free S-expression: Phase 2(c). Finally, the expanded S-expression is converted back to JavaScript syntax and we obtain a macro-free JavaScript program: Phase 2(d). The detail is described in the following subsections.

3.2 Parser Extension Phase

A major difficulty of implementing syntactic macro systems is the self-describing nature of a macro-enabled program whose syntax is described by the macro definitions that it contains. Separation of macro definitions from the rest of the program, in a sim-

¹<http://pegjs.majda.cz/>

ilar fashion to C header files, can partially alleviate the problem. However, such design decision precludes macro forms in the replacement template part of another macro definition, declaration of a macro having multiple patterns, and recursively defined macros.

Let's consider the previous example (Figure 5). The figure annotates the code regions with types of grammars used depending on the grammatical context of the program. Because the syntax of ExJS is an extension of JavaScript, it supports JavaScript syntax as well as user-defined macro syntax (**JS + Macro** in the body part of the figure). A macro definition consists of an array of “ $\langle pattern \rangle \Rightarrow \langle template \rangle$ ”. While the $\langle pattern \rangle$ is described by the pattern language of ExJS using its fixed syntax, the $\langle template \rangle$ is a JavaScript code, which in turn can contain macro usage, repeated patterns (e.g., $\langle pattern \rangle \dots$), and macro usage suffixes (**JS + Macro + Dots + Suffix**).

As figure 5 illustrates, the syntax of the macro-enabled language is context-dependent: the ExJS parser needs to switch the parser according to the context being parsed.

With this syntactic complication in mind, let us show our implementation technique of the extensible parser. The purpose of the parser extension phase is to generate a parser that recognizes the JavaScript syntax extended by the user-defined macros. Specifically, the syntax of the macro usage is defined by the pattern part of a macro definition. To resolve the self-describing nature of the macro syntax, the parser that is used for this phase ignores most of the code in the program except meta-variable declarations and macro patterns, which are defined in terms of a fixed syntax of the pattern language. The macro definition syntax separates a pattern and a template by $\Rightarrow$ and encloses them by a pair of braces. The use of $\Rightarrow$ and braces tell the parser the exact positions in the macro definition where the pattern and the template start and end. If either a pattern or a template contains an unclosed grouping symbol, the parser may fail to parse the macro declaration. This is the reason why ExJS requires for the usage of grouping symbols to be properly nested.

The 1st-stage parser skips over the macro templates and macro usages although the parser is ignorant of the extended syntax at this moment. This skipping is even more complicated by the existence of sometimes bothering JavaScript regular expression token syntax like “ $/\{/$ ” but PEG's ability to look into arbitrary number of characters ahead offers a plain and natural solution to such bothersome cases. We have appended to the set of parsing rules of the statement a rule that accepts arbitrary character. This rule has the least priority among the parsing rules of the **Statement** non-terminal symbol. When the parser comes to a point of a macro usage, all but the last parsing rule fails to match and the last one simply skips the first character of the macro usage. This process is repeated character-by-character manner until a recognizable form for the parser is found. The rollback involved in this process may appear costly but Packrat parsing technique [13] guarantees its linear computational complexity.

During the parser extension phase, the parser collects meta-variable declarations and macro patterns. The syntactic classes assigned to meta-variables are used to generate parsing rules for their occurrences in macro patterns. Macro patterns are used to generate parsing rules for macro usages.

Figure 6 presents the abstract syntax of the pattern language. The pattern part of a macro definition is a list of patterns which is either a literal, a meta-variable, a group of patterns, or a repeated pattern. ExJS supports all the literal types of JavaScript. A literal that occurs in a pattern matches an occurrence of the equivalent literal in a macro usage. Among four classes of meta-variables, **identifier**-, **expression**-, and **statement**-meta-variables correspond

```

pattern ::= Pat(name, ps)
ps ::= Nil | Pair(p, ps)
p ::= Lit(ltype, literal) | MVar(idtype, name)
    | Group(gtype, ps) | Repetition(p)
ltype ::= Null | Bool | Num | Str | Reg
idtype ::= Name | Expr | Stmt | Kwd
gtype ::= ( | { | [ | #

```

Figure 6: The abstract syntax of the macro pattern language

```

Tps[Nil] = ε, Tps[Pair(p, ps)] = e __ es
Tp[Lit(ltype, literal)] = v: ltype
    &{ return eval(v.value) ≡ literal; }
Tp[MVar(Name, _)] = MacroIdentifier
Tp[MVar(Expr, _)] = AssignmentExpression
Tp[MVar(Stmt, _)] = Statement
Tp[MVar(Kwd, kwd)] = v: MacroKeyword
    &{ return v.name == "kwd"; }
Tp[Group( (, ps)] = " ( __ es __ )"
Tp[Group( [ #, ps)] = es
Tp[Repetition(p)] =
    In program → (e ( __ e) *) ?
    In template → (e ( __ e) * ( __ " . . ." ) ? ) ?
e and es stand for Tp[p] and Tps[ps], respectively

```

Figure 7: A set of conversion rules that translates abstract syntactic nodes to PEG rule representations

to the syntactic domains of identifiers, expressions, and statements of JavaScript, respectively. A **keyword** meta-variable matches an occurrence of identifiers that literally are equivalent with the name of the meta-variable and is converted to a string literal, internally. Four types of grouping symbols, namely parentheses, braces, brackets, and phantom-groups, package a list of patterns into a compound pattern.

Once the syntax of macro patterns are examined, two PEG grammars are generated. One is used to parse macro forms that appear in the program context (e.g., **JS + Macro**) and the other is used to parse the template part of the macro definition (e.g., **JS + Macro + Dots + Suffix**). Figure 7 shows the conversion rules from macro patterns to the corresponding PEG patterns. A list of patterns is converted to a sequence of PEG non-terminal symbols that correspond to those patterns. The symbol “__” detects white spaces. Five kinds of Literals (Lit(ltype, literal)) are converted to respective non-terminal symbols of ECMAScript grammar. The side condition of the literal rule guarantees that the recognized literal symbol is value-equal to the one specified in the macro definition following the specification of `syntax-rules` macro. This feature allows a string pattern “str” to match against another equal but not identical string styles such as ‘str’ and ‘\x73\x74\x72’. Three types of meta-variables, namely Name, Expr, and Stmt, that occur in a macro pattern are converted to respective PEG symbols. In contrast, a **keyword** meta-variable is converted to MacroKeyword symbol which accepts a valid JavaScript identifier or a sequence of symbols that are string-equal to the given kwd. A group of patterns are converted to a sequence of PEG expressions enclosed in a pair

Table 1: Conversion from macro-enabled JavaScript to S-expression

JS (c, v, e, s)	S-expression (V, S)
var $v = e$	(define $V_v S_e$)
v	V_v
StringLiteral str	("JS" "string" " str ")
c	("JS" "const" " c ")
[$e, \dots$]	("JS" "array" $S_e \dots$)
$e_1 \oplus e_2$	("JS" "b-op" " $\oplus$ " $S_1 S_2$)
[$\#e \dots \#$]	($S_e \dots$)
function $f(v) \{s\}$	(define ($V_f V_v$) S_s)
function (v) $\{s\}$	(lambda (V_v) S_s)
if (e) s	("JS" "if" $S_e S_s$)
M $e \dots$	($V_M S_e \dots$)
$f(e)$	("JS" "funcCall" $V_f S_e$)

of grouping symbols if the pair of grouping symbols that occur in the pattern are either parentheses, braces, or brackets. If they are phantom-grouping ([$\#$ and $\#$]), PEG patterns for grouping symbols are not generated. Repetition patterns (e.g., “ $\langle pat \rangle \dots$ ”) are context-dependent patterns. If it occurs in a program context (outside the macro definitions), it is converted to a sequence of zero or more patterns. If it occurs in a template context, it is converted to a similar PEG expression that allows an optional dots notation ($\dots$) to appear as its suffix.

AssignmentExpression, which is a non-terminal PEG symbol that corresponds to the left-expression of JavaScript, is used as the target for an **expression** meta-variable, instead of more general Expression that stands for right-expressions. The design decision is made by our favor to use commas as punctuations in the macro forms. If we let expression meta-variables to be converted to Expression, a usage of **let** macro such as “**let** $a = 1, b = 2$ **In** $\dots$ ” would be interpreted as an occurrence of an application of a comma operator inside the **let** form: “**let** $a = (1, b = 2)$ **In** $\dots$ ”. This **let** code is parsed this way because the parsing rule for the macro form is given lower precedence than the rules for the standard JavaScript forms.

We have used a PEG.js’s non-standard feature called the *semantic predicate* ($\&\{guard\}$) to implement context-dependency and equality testing over literals and keywords. This feature allows us to add to a PEG parsing rule a conditional guard that can contain arbitrary JavaScript code. To implement context-dependent grammar rules, we have prepared a global data structure, which saves the context information during parsing and is referenced from a few context-dependent PEG rules. Although we found semantic predicates useful, they can be replaced by standard PEG mechanisms.

Now we have generated PEG descriptions that describe the extended syntax introduced by macro definitions found in a macro-enabled program. We obtain the complete PEG descriptions for macro-extended syntax by integrating them with the PEG description of the standard JavaScript syntax. This completed grammar description is passed to PEG.js to generate the 2nd-stage parser for the second phase.

3.3 Macro Expansion Phase

As explained earlier, the proposed macro expansion process relies on mutual conversion between JavaScript and S-expression. In this mutual conversion scheme, there are two important issues. Firstly, variable bindings must be maintained so that hygienic variable renaming system implemented by R⁶RS can recognize the variable bindings and usages in the transformed program. Secondly the conversion process should not change the meaning of the values

Table 2: Functionality

Functionality	LOC
Macro definition syntax (PEG)	595
Phase 1 (JS)	689
Conversion of AST: JSON $\rightarrow$ S-exp	208
Expansion: S-exp $\rightarrow$ JS (Ypsilon)	439
Miscellaneous scripts	
Total	1,974

and code of the program.

For this purpose, we chose for the conversion target for an ExJS program an S-expression that represents variable bindings of the program. On the other hand, entities in the program that are unrelated to variable bindings such as constant values, data structures, operators, control structures, and function applications are all represented by tagged data structure so that they are not manipulated by the macro expander and convey enough information for inverse transformation.

Table 1 illustrates our conversion scheme. The variable declaration (using **var**), the variable reference (v), and named/anonymous function abstraction (using **function**) are represented by corresponding Scheme forms, **define** special form, variable reference, and **lambda** abstraction, respectively, while arrays, arithmetics, control structure, and function applications are represented by tagged data structure in such a form as “(“JS” “kind” $\dots$)”. Constant values are packaged in Scheme strings. As a macro usage (i.e., (**M** $e \dots$)) is the target of the macro expander, it is translated to a Scheme-style macro usage by grouping in a pair of parentheses. Figure 8 (b) presents the S-expression that is converted from an ExJS code for the **let** example (see also Figure 2).

Such obtained S-expression is expanded by the macro expander of an R⁶RS Scheme interpreter, in our case Ypsilon.² Now the macro definition has been removed and an occurrence of the **let** macro usage has been expanded according to the corresponding replacement template and is replaced by a lambda application. It should be noted that the hygienic macro expander has renamed the variable introduced by the macro template, namely $V-a$ and $V-b$ to fresh names $V-where_200_$ and $V-a_200_$, respectively.

The final step of the macro expansion process is to convert the macro-free S-expression to JavaScript. See Figure 9 for the result of unparsing. Because hygienic macro expansion has removed all the macro related elements from the S-expression, this conversion step is just a simple unparsing. Variables that were given exotic names by Ypsilon’s macro expander during the variable renaming process are renamed properly to one that JavaScript accepts as valid identifiers.

3.4 Current Implementation

Current implementation of ExJS is written in about 2,000 lines of code using PEG, JavaScript, and Scheme as shown in Table 2. We have reused David Majda’s PEG.js-based JavaScript parser that conforms to full ECMA-262 (5th edition) standard specification [11]. The first item in the table stands for the portion of the PEG grammar definition for the **expression** and **statement** macro declaration syntax, which are inserted into Majda’s PEG description of JavaScript grammar as top-level declaration forms. Parser extension part (phase 1) is implemented by JavaScript. It includes analysis of the pattern definitions, and generation of template PEG grammars for the user-defined macro forms. The data representation of the abstract syntax tree of Majda’s parser is in JSON form and so is our parser implementation. As the standard R⁶RS does

²<http://code.google.com/p/ypsilon>

```

(begin
  (define-syntax let-Macro
    (syntax-rules
      (V= V-In)
      ((_ (V-v V-= V-e)
        ... V-In V-body)
        ("JS" "funcCall"
          ("JS" "function" "#\nul
            (lambda (V-v ...)
              ("JS" "return" V-body) #@))
            V-e ...))))))
  (begin
    (define V-where ("JS" "string" "global"))
    (define V-a ("JS" "const" "1"))
    (define V-b ("JS" "const" "2"))
    (let-Macro
      (V-where V-= ("JS" "string" "local"))
      (V-a V-= ("JS" "const" "100"))
      V-In
      ("JS" "funcCall"
        ("JS" "propAccess" V-console "log")
        V-where V-a V-b))
    ("JS" "funcCall"
      ("JS" "propAccess" V-console "log")
      V-where V-a V-b))

```

(a)

```

(begin
  (define V-where ("JS" "string" "global"))
  (define V-a ("JS" "const" "1"))
  (define V-b ("JS" "const" "2"))
  ("JS" "funcCall"
    ("JS" "function" "#\nul
      (lambda (V-where_200_ V-a_200_)
        ("JS" "return"
          ("JS" "funcCall"
            ("JS" "propAccess" V-console "log")
            V-where_200_ V-a_200_ V-b))
          #@))
    ("JS" "string" "local")
    ("JS" "const" "100"))
  ("JS" "funcCall"
    ("JS" "propAccess" V-console "log")
    V-where V-a V-b))

```

(b)

Figure 8: The Scheme interpreter expands the macro-enhanced S-expression (a) and generates a macro-free S-expression (b)

```

var where = "global";
var a = 1;
var b = 2;
((function (where_200_, a_200_) {
  return
    ((console.log) (where_200_, a_200_, b));
}) ("local", 100));
((console.log) (where, a, b));

```

Figure 9: Unparsing the macro-free S-expression generates a macro-free JavaScript

```

expression I_travel {
  symbol: A, B, means;
  keyword: from, to, by;
  { I_travel [# from A to B by means #], ... =>
    [{ from: A, to: B, by: means }, ...] }
}

```

```

var path = I_travel
  from Home to TokyoNarita by Bus,
  from Narita to Vancouver by IntlFlight,
  from Vancouver to MontrealTrudeau by Domestic,
  from MontrealTrudeau to UnivMontreal by Taxi;

```

Figure 10: Using the symbol meta-variables, **I_travel** macro usages turns into a form resembling natural language.

not offer a JSON library, we relied on Racket³ [12] to convert the abstract syntax tree represented in JSON format to S-expression format. The macro expansion of the program represented in S-expression format is done by the Ypsilon Scheme interpreter. The resulting macro-free S-expression is finally unparsed to generate JavaScript code. Most of the Ypsilon code is for unparsing.

ExJS is a source-to-source macro expander for JavaScript. The user writes a macro-enhanced JavaScript, runs ExJS to expand macro definitions and usages in the source code, and obtains a macro-free JavaScript code.

4. BELLS AND WHISTLES

In sections 2, we presented the core feature of ExJS. This section explains the additional features that were added to ExJS to improve its expressiveness.

4.1 Symbols

Symbols are the first class citizen in the LISP family of languages but they are absent from most other programming languages. We have added support for symbols for a programming language that lacks first class support of symbols, by introduction of a new meta-variable class called **symbol**. As far as parsing is concerned, a **symbol** meta-variable and an **identifier** meta-variable are treated in the same manner. Any JavaScript identifier matches with either of them. When a JavaScript identifier matches with a **symbol** meta-variable, then it is internally converted to a string literal. This feature effectively allows the macro programmer to remove quotations from the string data and leads to more natural description of macro usages.

Figure 10 exemplifies the use of symbol meta-variables. The **I_travel** is a macro to describe a JSON data structure that represents a plan to travel from one place to another. We could write a similar macro using **expression** meta-variables in places of symbol meta-variables. However in that case, the macro form must contain valid JavaScript expressions where unquoted symbols like *Narita* are disallowed. In the example, because the three meta-variables, *A*, *B*, and *means*, are declared as **symbols**, symbols such as *TokyoNarita* and *MontrealTrudeau* are converted to JavaScript string values. The following is the result of macro expansion process for the code given in the figure.

```

var path = [
  { from:"Home", to:"TokyoNarita", by:"Bus" }, ...
  { from:"MontrealTrudeau", to:"UnivMontreal",
    by:"Taxi" } ];

```

³<http://racket-lang.org/>

In the **I_travel** example, we have used the camel-case style to concatenate multiple terms into a single symbol. Because the alphabet of JavaScript identifiers includes UTF8 characters, use of oriental alphabet symbols are allowed for ExJS symbol. This is especially useful for east-Asian programmers as words in a sentence are not separated by white spaces in a natural east-Asian languages (including Japanese, Korean, and Chinese), regardless of the phrase length. When this feature is used in the macro usage, the ExJS program appears as a natural Japanese description of my travel plan directly embedded in a JavaScript program.

```
var path = 私の旅程は
    東京 から 成田 まで バス を使い
...
Montreal の Trudeau 空港 から Montreal 大学 まで
タクシー を使います.
```

4.2 Punctuation

The examples that we have seen so far implicitly used punctuation characters such as commas and semi-colons. All of the punctuation characters that are supported by JavaScript are recognized by the macro pattern language as well.

Sometimes we would like to use more complex symbols for separating phrases. For example, our macro definition uses `=>` to separate the pattern and the template part. For another example, OCaml's **let** syntax separates a set of variable initializations using the symbolic punctuator **and**.

ExJS supports symbolic punctuators (e.g., `=>`), alphabetical punctuators (e.g., **and**), and arbitrary sequences of punctuators (e.g., **such that**).

Punctuators frequently occur in a repetition pattern. For example, OCaml-style **let** syntax can be defined by the following ExJS pattern.

```
let vl = el and ... In e
```

In this example, the initialization pattern, `vl = el`, as well as **and** are repeating in the macro pattern. To accommodate this issue, the PEG translation rule for the repetition pattern is slightly modified as follows.

For program context:

```
(e ( __ es __ e ) * ) ?
```

For template context:

```
(e ( __ es __ e ) * ( __ es __ "..." ) ? ) ?
```

In these patterns, an occurrence of *e* denotes the repeated pattern and *es* stands for a sequence of punctuations symbols. In Figure 6, the abstract syntax for the repetition pattern has been presented as **Repetition**(*p*) but to incorporate this punctuation feature, it is modified to **Repetition**(*p*, *ps*) so that it remembers the sequence of punctuations placed between the repeated pattern and the dots.

5. DISCUSSION

5.1 Performance of Macro Expansion

As we have explained in subsection 3.4, all the features of ExJS including those described in the last section, are implemented in less than 2,000 lines of code of PEG, JavaScript, and Scheme. The implementation of our previous work [2] is done in about the same lines of code of Scheme. However, in this work the implementation of the parser relied on an ad-hoc extension of a top-down operator precedence parser and that implementation suffered from

poor extensibility, was difficult to understand, and posed restricted meta-syntax. On the other hand, abstraction layers in the current implementation is cleanly separated and easy to understand, and gives much greater degree of freedom to the macro program to define macro syntax.

A weakness with ExJS is its somewhat slow parser generator. From a series of benchmark conducted on an iMac (Intel Core 2 Duo, 2.93Ghz, 8GB 1067MHz DDR3 memory, Mac OS X 10.9.3), macro expansion of an empty JavaScript program takes about 1.57 seconds. Most of the time is consumed in generation of the second phase parser. As an empty program defines no macro, this is the time required to generate a parser that conforms to JavaScript syntax.

An earlier implementation of ExJS occasionally demonstrated performance degradation for parser generation. For example, as for the **I_travel** macro, PEG.js took about 7 seconds to generate a parser from the PEG grammar created by ExJS from macro definition. It was observed that PEG.js takes time proportional to exponential of the PEG group nesting level. Currently, this performance bug is addressed by avoiding nested use of PEG groups.

To address the problem of parser generation time, we have added a *parser cache* capability to the ExJS macro processor. ExJS's parser cache consists of macro patterns and corresponding parsers generated by PEG.js. Parser generation is executed only when ExJS recognized unknown macro patterns in the macro definition. As far as the shape of the macro does not change, the parser extension phase skips parser generation, and simply loads the prebuilt parser from the cache.

Except for parser generation cost, time required for macro expansion is negligible.

5.2 Support for Other Languages

We are currently trying to apply the proposed framework to implementation of macro systems for a few other programming languages that include Scala, Java, and C. The success of ExJS is partly supported by language features of JavaScript such as arbitrarily nestable scopes and the support of first-class closures. They are used to limit the scope of macro-injected variables and the latter is. An implementation of a hygienic macro system for Scala have been done without much trouble based on the proposed approach by Kensuke Mori [19]. This experience suggests good portability of the proposal to functional programming languages. Java and C, on the other hand, are interesting languages as they do not offer first-class closures.⁴ We are currently seeking a way to extend the proposed framework by adding a language-portable technique to address the difficulty.

5.3 Interference with the Precedence Rule

The reader may have wondered why the **let** example uses the awkward capitalized keyword “**In**” in Figure 2. If we replace **In** with more natural **in** a problem arises, due to precedence between the binary property-inclusion operator and the macro keyword. Because the parse rule of a macro usage in ExJS is assigned the lowest precedence in the expression syntactic class, the parser treats **in** in a **let** macro usage as a binary operator. Consequently, any seemingly valid usage of the **let** macro fails.

On the contrary, if we give the highest precedence to macro usages in the expression syntactic class, we sometimes face with unpleasant case when a binary **in** operator appears in a usage of the **let** macro, such as the following.

⁴The supports of closures in C++11 and Java 8 are limited and insufficient for our framework.


```

1  let haveField = x in object
2  in if (haveField) ...

```

In this example the two occurrences of **in** are intended as a binary operator and the keyword of the **let** macro, respectively. However, if the macro usage is given a higher precedence, ExJS parser assumes the first occurrence of **in** to be the macro keyword and the second to be a binary operator, and raises a parse error. We can easily fix this problem by surrounding the usage of the binary **in** operator as follows:

```

1  let haveField = (x in object)
2  in if (haveField) ...

```

Although, the source of this problem is the grammar ambiguity resulting from using the same keyword for different purposes, the error diagnostics raised by the PEG-based parser generated by ExJS is impenetrable. Culpepper and Felleisen proposes an elegant macro system called *syntax-parse* that allows the user to define robust macros from easy-to-understand specification.[8]

6. RELATED WORK

To the extent of the authors' knowledge, the first non-LISP (programming languages with richer, more conventional syntax) that incorporated hygienic macro system is the object-oriented programming language Dylan [23]. Dylan allows the macro programmer to introduce new syntactic forms that are used in the definition, statement, and expression contexts. The implementation scheme described in [3] internally processes macro definitions by pattern matching against a set of fixed meta-syntax called *skeletons*. This is the reason for the rather limited syntactical extensibility of Dylan.

Honu attempts to add an algebraic syntactic layer to Racket and supports Racket-style hygienic macro system.[21] Its software architecture relies on hooks offered by Racket to its lexing/parsing pipeline. It would be difficult to adopt a similar technique in other programming language implementations, which in general lack such features.

Our previous work [2] and *sweet.js*⁵ are two hygienic macro systems for JavaScript. Both systems support macro features similar to R⁶RS *syntax-rules*, such as pattern match and dots. In comparison, the proposed approach has greatly simplified the software architecture of the extensible parser and it also decoupled the implementation of the macro expander from the implementation of the macro system. In consequence, this new implementation framework can be easily replicated on other programming languages.

Both Fortress [1] and ExJS use PEG-based, staged-parsing techniques. The benefit of their PEG-based implementation strategy is its simplicity but it also introduces difficulty in incorporating sophisticated macro features such as *macro-defining-macros*. It seems, however, that it is possible to support such macro definitions via multi-staged parsing technique. The macro expander of Fortress is tightly coupled with its implementation but ExJS delegates the task of macro expansion to Scheme. The strength of our light-weight implementation framework is its general versatility in the sense that it successfully decouples the implementation of macro expander from the implementation of the host programming language.

There are compile-time meta systems that are sometimes categorized as syntactic macro systems [9, 24, 25]. They offer an API to extend the parser and enable the programmer to access the abstract syntax tree. Although such API offers full flexibility, this

⁵<http://sweetjs.org/>

style of compile-time meta programming requires the programmer the detailed knowledge of the language implementation. Because the API is dependent on specific implementation of the language, the resulting meta-program is not portable to other implementation of the same language. In comparison with Camlp5's complex API that offers 42 types of abstract tree nodes, where each of which is an abstract data type offering dozens of data constructors [9], the skeleton macro systems incorporated in Scheme, Dylan, *sweet.js*, and ExJS are much simpler and easier to learn.

Nemerle [25] offers a hygienic macro system but its default macro form syntax is the same as a function application form. Nemerle also allows the macro programmer to define arbitrary macro syntax but for that purpose the programmer has to declare the syntax in terms of token sequence. In contrast, ExJS's imposes minimal requests that macro syntax starts from its name and the use of grouping symbols must properly nested. It also offers repeated macro patterns through the dots syntax (*. . .*) and is capable of defining recursively defined macros which other hygienic macro systems do not support.

7. SUMMARY

ExJS is an extension of JavaScript that offers syntactic extensibility of the core language through hygienic syntactic macro facility. The functionality of the hygienic macro system of ExJS is similar to *syntax-rules* macro system of R⁶RS Scheme. The **expression**- and **statement**-macros are defined by a simple pattern matching. The pattern language supports *repeated patterns* using the *dots*-notation (*. . .*). We have identified mismatch between the *pattern-part* (the part in the macro definition that defines the structure of the macro form) and the *template-part* (the part that defines how the macro form should be expanded to) with respect to treatment of the repeated patterns. To deal with this problem, we have introduced the notion of *suffix pattern*. Suffix patterns allow the programmer to define recursively defined macro forms.

The implementation of a flexible syntactic macro system is complicated because it needs to deal with self-defined syntax and requires parser extensibility. We believe these are causes that prevented incorporation of flexible macro system to programming languages with conventional syntax. We addressed this difficulty by multi-stage parsing technique, and generation of context-dependent syntax rules, using the parsing expression grammar technology.

The proposed implementation scheme successfully decouples the implementation of macro expander from the implementation of the extensible parser of the host programming language. As the task of macro expansion is entirely delegated to R⁶RS Scheme, the technical difficulty of implementing a hygienic macro system for general programming languages is considerably reduced. Additionally, the implementation of the extensible parser has been greatly simplified by incorporation of the PEG-based parsing technique. As the result, the ExJS macro processor has been implemented in less than 2,000 lines of JavaScript and Scheme.

The proposed software architecture for hygienic macro systems is general and can be easily reused to build a macro system for another programming language. There are features that are available in modern hygienic macro system, such as *syntax-case* macro definition, local macros, *macro-defining-macros*, and location mapping, but they are not offered by our current implementation. Although we believe some of them can be addressed with certain effort,⁶ to what extent the proposed implementation frame-

⁶We believe local macros can be added by introduction of context sensitiveness to express locality using the technique. We also anticipate *macro-defining-macros* is realized by incorporation of a

work can cover remains further investigation.

Acknowledgment.

We gratefully acknowledge the anonymous ILC 2014 reviewers who gave generously of their time and share their careful editing and comments about a draft of this paper. This work was partially supported by JSPS KAKENHI Grant Numbers 23500034, 23700043, and 26330079.

8. REFERENCES

- [1] E. Allen, R. Culpepper, J. D. Nielsen, J. Rafkind, and S. Ryu. Growing a syntax. In *Proceedings of Workshop on Foundations of Object-Oriented Languages*, 2009.
- [2] H. Arai and K. Wakita. An implementation of a hygienic syntactic macro system for javascript: a preliminary report. In *Workshop on Self-Sustaining Systems, S3*, pages 30–40. ACM, 2010.
- [3] J. Bachrach and K. Playford. D-Expressions: Lisp power, Dylan style. Nov. 1999.
- [4] A. Bawden and J. Rees. Syntactic closures. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 86–95, New York, NY, USA, 1988. ACM.
- [5] W. Clinger and J. Rees. Macros that work. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 155–162, New York, NY, USA, 1991. ACM.
- [6] W. Clinger and J. Rees. *Revised⁴ report on the algorithmic language Scheme*, nov 1991.
- [7] D. Crockford. *Beautiful Code: Leading Programmers Explain How They Think*, chapter 9: Top down operator precedence, pages 129–145. Theory In Practice. O'Reilly, June 2007.
- [8] R. Culpepper and M. Felleisen. Fortifying macros. In *in proceedings of the international conference on Functional programming*, Baltimore, USA, Sept. 2010.
- [9] D. de Rauglaudre. *Camlp5 – Reference Manual*. Institut National de Recherche en Informatique et Automatique, version 6.00 edition, Oct. 2010.
- [10] R. K. Dybvig, R. Hieb, and C. Bruggeman. Syntactic abstraction in scheme. *LISP and Symbolic Computation*, 5(4):295–326, 1992.
- [11] ECMA Standardizing Information and Communication Systems. *ECMAScript language specification (Standard ECMA-262)*. ECMA International, final draft standard ECMA-262, 5rd edition edition, Sept. 2009.
- [12] M. Flatt, R. B. Findler, and J. Clements. GUI: Racket graphics toolkit. Technical Report PLT-TR-2010-3, PLT Inc., 2010. <http://racket-lang.org/tr3/>.
- [13] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 7th ACM SIGPLAN international conference on functional programming (ICFP '02)*, pages 36–47, Oct. 2002.
- [14] B. Ford. Parsing expression grammars: A recognition-based syntactic foundation. In *Proceedings of the 31st annual ACM SIGPLAN - SIGACT symposium on principles of programming languages*, pages 111–122, Jan. 2004.
- [15] B. Kernighan and D. M. Ritchie. *The C programming language*. Prentice Hall, second edition edition, 1988.
- [16] D. E. Knuth. *T_EX: The Program*, volume Volume B of *Computers & Typesetting*. Addison-Wesley Professional, Jan. 1986.
- [17] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161, New York, NY, USA, 1986. ACM.
- [18] B. Leavenworth. Syntax macros and extended translation. *Communications of the ACM*, 9(11):790–793, 1966.
- [19] K. Mori. An implementation of a hygienic macro system for Scala using a Scheme macro expander (in japanese). Master's thesis, Department of Mathematical and Computing Science, Tokyo Institute of Technology, Feb. 2014.
- [20] V. R. Pratt. Top down operator precedence. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 41–51, New York, NY, USA, 1973. ACM.
- [21] J. Raffkind and M. Flatt. Honu: syntactic extension for algebraic notation through enforestation. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, pages 122–131. ACM, 2012.
- [22] R. Seindal, F. Pinard, G. V. Vaughan, and E. Blake. *GNU m4 manual*. GNU, 1.4.16 edition, 2011.
- [23] A. Shalit, O. Starbuck, and D. Moon. *The Dylan Reference Manual*. Addison-Wesley Developers Press, Sept. 1996.
- [24] T. Sheard and S. Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.
- [25] K. Skalski, M. Moskal, and P. Olszta. Meta-programming in Nemerle. In *Generative Programming and Component Engineering*, 2004.
- [26] G. L. Steele, Jr. *Common Lisp: the Language*. Digital Press, Newton, MA, USA, second edition, 1990.
- [27] D. Weise and R. Crew. Programmable syntax macros. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 156–165, New York, NY, USA, 1993. ACM.

multi-staged parsing technique that lets ExJS to repeat parser extension phase until the code becomes absent from macro definitions.