# An Implementation of A Hygienic Syntactic Macro System for JavaScript: A Preliminary Report

Hiroshi Arai
Department of Mathematical and Computing Science, Tokyo Institute of Technology
harai6@is.titech.ac.jp

Ken Wakita
Department of Mathematical and Computing Science, Tokyo Institute of Technology
wakita@is.titech.ac.jp

## Abstract

The article describes an implementation scheme of a hygienic syntactic macro system for JavaScript. Instead of implementing the complex logic of a hygienic macro system from scratch, the proposed method heavily relies on an existing Scheme implementation of its hygienic syntactic macro system. A program written in our macro-enhanced version of JavaScript is first translated into a Scheme program. It is then macro-expanded by a macro expander of Scheme into a macro-free Scheme code. Finally, it is translated back to Javascript, which at this point is free of macros. To deal with the macro-enhanced syntax, an extensible parser architecture based on top-down operator precedence is proposed. A prototype hygienic macro system, including both the parser and the two-way translator, is implemented by only 2,000 lines of Scheme code.

***Keywords*** Hygienic macro system, Parser, Top-down operator precedence, Program transformation, JavaScript, Scheme

## 1. Introduction

A *macro* system adds syntactic extensibility to a programming language, which has a fixed syntax that is otherwise inextensible. Using a macro system, programmers can introduce new syntactic elements to the programming language. Using the extended syntax thus defined, you can enjoy a higher-level, abstract, declarative, and robust style of programming.

Macro uses found in a program are expanded by a *macro transformer*, or a *macro expander*, into code fragments written in the macro-free subset of the programming language. Because programs that are expanded by the macro transformer do not contain macro definitions or macro uses, they are understandable by the unextended subset of the programming language. Hereafter, we will call the subset of a programming language which does not offer macro capabilities as *core programming language*, in contrast to *full programming language* that includes macro-based extensibility. Also, we will call the whole system that processes programs of the full programming language as *full system* and the one that processes the base subset as *core system*.

Some macro systems, called *lexical macro systems*, such as Unix `m4` [21] command and *C preprocessor* (*CPP*) [16], perform simple text-based substitution over text. Because lexical macro systems are ignorant of the grammar of the core programming language, they sometimes perform unexpected program expansion and generate ill-formed programs.

*Syntactic macro systems* are those that are integrated with the core programming language (e.g., Common Lisp [23], Scheme [2], Dylan [22], camlp4 [11]) and are aware of the syntactic structures of the core programming languages. Syntactic macro systems can be regarded as tree transformers that operate on abstract syntax trees of programs. For this reason, it is easier to write syntactically correct macros.

Hygienic syntactic macro systems [3, 7, 12, 18, 22], first proposed for the programming language Scheme, are called so because of their sound treatment of variable bindings. They pay close attention to variable bindings in the context of macro usage and carefully avoid accidental name collisions between temporary variables used inside a macro definition and those that are bound in the lexical environment.

In spite of their soundness, power, and usefulness, hygienic syntactic macro systems have not been adopted by most programming languages; we can see their implementations only in various Scheme implementations and a few research prototypes of other languages.

There are two technical obstacles for introducing hygienic syntactic macro systems to general programming languages. Firstly, syntactic extension is much harder to implement for non-LISP languages than for LISP and its variants.

Because LISP is based on the "programs as data" philosophy, its forms, including macro forms and forms expanded from them, are represented all by a data structure called *S-expression*s. We can therefore use a standard S-expression parser to handle macro-extended LISP forms. In contrast, even if we wanted to add a new construct, **try** ... **catch** ... to C, for example, the standard parser of the C language would not be able to recognize it. In other words, to add a syntactic macro system to a non-LISP language, its parser is required to be extendable in order for it to be able to deal with user-defined syntactic elements.

The other problem is due to the software architecture of the "*portable*" implementation of hygienic macro system for Scheme [14]. This portable implementation is bootstrapped by using Scheme's hygienic macro. This bootstrapping technique adopted in the portable implementation makes it hard for it to be ported to another programming system if it does not natively support hygienic macros.

Our goal is to identify a universal method to port a hygienic macro system to non-LISP languages such as Java, Haskell, JavaScript, and others. In this article, we will report our first attempt, with JavaScript as the target language. Two contributions of our proposal are an extensible parsing architecture based on *top-down operator precedence* and a macro expander that piggybacks on a macro expander implemented in and for Scheme.

Our strategy is summarized as follows: An extensible parser takes a macro-enhanced JavaScript program and builds its abstract syntax tree. When the parser recognizes a macro definition, it modifies itself so it can handle future uses of the macro. Our system contains a two-way converter between JavaScript and Scheme. First, the JavaScript-to-Scheme converter converts the abstract syntax tree of a macro-enhanced JavaScript program to an S-expression, which is macro-expanded by a Scheme macro expander. Finally, the macro-free S-expression obtained from macro expansion on the Scheme side is converted back to JavaScript by the reverse converter.

Currently, JavaScript is the only programming language that our prototype system supports, but hopefully it will serve as a template when you want to add a hygienic macro system to a different programming language.

The rest of the paper is organized as follows: Section 2 reviews the existing macro systems. Section 3 overviews our implementation scheme, whose extensible parser and macro expander are explained in detail in Section 4 and Section 5, respectively. Section 6 shows a few illustrative usages of our macro system. Section 7 compares our proposal with related work, and Section 8 concludes the paper.

## 2. Macro Systems

This section reviews a few classes of macro systems, starting from naive lexical macro systems to syntactic, hygienic macro systems (see Table 1 for the outline). In this review, we will show the high-level descriptive power of pattern-match-based syntactic macros and semantic soundness guaranteed by hygienic syntactic macro systems.

### 2.1 Lexical Macro Systems

A macro expander of a *lexical macro system* simply replaces a part of text that matches a macro form with its definition. A lexical macro can be parametrized by macro arguments. The following is an example of macro definitions written in *CPP* (C Pre-processor) [16], which is arguably the most known lexical macro system.

```
#define Square(x) ((x) * (x))
```

The Square macro is a parametrized macro and takes an argument denoted by $x$. The definition comprises a *macro pattern* part, Square($x$), and an *expansion* part, (($x$) * $x$)). The CPP macro expander detects a use of the macro, such as "`Square(3 + 4)`", and replaces it, according to the expansion part of the macro, with "`((3 + 4) * (3 + 4)`". Macros are expanded before compilation takes place. Programmers can thus manipulate the source program according to the compilation environment using conditional macro declarations and also can use the macro expander as a handmade partial evaluator. Because the CPP macro system is untyped, unlike its strongly typed core language C, the CPP macro system can be used to add certain types of polymorphic abstraction to the core language.

Simple and often useful lexical macros sometimes become sources of subtle, hard-to-catch programming bugs. For example, let us see the following slight modification of the Square macro.

```
#define Square2(x) x * x
```

The CPP macro expander expands a program fragment which uses the macro, "`2 * Square2(3 + 4)`" to "`2 * 3 + 4 * 3 + 4`", which is not what programmers would normally expect. There are a couple of problems here. One is that the lexical macro system is unaware of the syntactic structure of the program being macro-expanded. A compound expression, such as "`3 + 4`", that the macro receives as its parameter can be split into two sub-expressions, "`2 * 3`" and "`4 * 3`," when this expression is processed by the C compiler. Another problem is that a part of the expansion of the macro form can be split from the rest and merge into its surrounding context to make a sub-expression, like "`2 * 3`".

Other well-known lexical macro systems are `m4` [21] and TEX[17]. The former is a general purpose macro system, in that it is not targeted for a specific programming language. The m4's macro expander supports powerful text manipulation features including regular expressions. A large part of the functionality of the document preparation system TEX and its extension LATEX is built on top of TEX's macro system.

**Table 1.** Comparison of macro systems: Curious readers are referred to Brabrand and Schwartzbach's extensive survey [4].

| | Level of operation | Programmability | Hygienic | Core language |
|---|---|---|---|---|
| CPP [16] | lexical | conditionals | no | C |
| m4 [21] | lexical | arithmetic | no | generic |
| TEX [17] | lexical | yes | no | TEX |
| Camlp4 [11] | sytanctic | yes | no | Objective Caml |
| MetaBorg [6] | sytanctic | yes | no | generic |
| Common Lisp [23] | sytanctic | yes | no | Common Lisp |
| Scheme [2] | sytanctic | yes | yes | Scheme |
| Dylan [22] | hybrid | no | yes | Dylan |
| MS$^2$ [25] | sytanctic | yes | no | generic |
| C++ Template [24] | sytanctic | constant folding | no | C++ |
| Our Method | sytanctic | no | yes | generic |

### 2.2 Syntactic Macro Systems

A macro system is called *syntactic* if it is aware of the syntactic structure of its core language [19]. Unlike lexical macro systems, which operate on lexical tokens of the input program, syntactic macro systems manipulate the abstract syntax tree; macro variables are bound to sub-trees in the abstract syntax tree and a syntactic macro expander substitutes a subtree that matches a macro pattern form with the one that corresponds to its expansion form. Because of this syntax awareness, syntactic macro systems are free from the problem we stated earlier regarding lexical macro systems.

Programming languages of the Lisp family support a kind of syntactic macro system called `defmacro`. Because a LISP program is expressed in terms of its basic data structure called *S-expression*, LISP can manipulate its program as data and later evaluate it in the course of its execution. Using this facility for meta-programming, a LISP programmer can write arbitrary macros.

```
(defmacro Square (X)
  (list '* X X))
```

When a LISP expression, "`(* 2 (Square (+ 3 4))`" is passed to the macro expander, the macro argument "`(+ 3 4)`" is bound to the macro variable, *X*, and is expanded to the following expression.

```
(* 2 (* (+ 3 4) (+ 3 4)))
```

Unlike the Square2 macro for the CPP example, macro expansion for `defmacro` preserves the syntactic structure of the macro use, in that the subexpression bound to the macro variable is not broken into pieces or interleaved with its surrounding context.

The use of explicit program manipulation in the definition of the Square macro may seem cumbersome and complicated. By using `quasi-quote` (the back-quote operator) and `quasi-unquote` (the comma operator), we enjoy the pattern-matching flavor of the CPP macro system:

```
(defmacro Square (X)
  `(* ,X ,X))
```

*Camlp4* [11] is a syntactic macro system for Objective Caml and offers a preprocessor and a pretty-printer. It makes the abstract syntax trees of programs written in its core language accessible to the macro writer. It defines data structures that model abstract syntax trees and offers APIs to infer about and modify abstract syntax trees.

Because camlp4 does not offer a high-level pattern language, macro writers need to learn camlp4's data structures and APIs in addition to the syntax of the Objective Caml language. Additionally, because the syntax of Objective Caml is far more complicated than that of LISP[1], the camlp4 macro system is away harder to use than `defmacro`.

*MetaBorg* [6] offers a toolkit to build a syntactic macro system targeted for arbitrary programming languages. A macro writer using MetaBorg first needs to build a parser of the core language using MetaBorg's parser generator. He then defines the syntax of his macro, which is regarded as a patch to the syntactic definition of the core language. He also writes a rule to manipulate the macro syntax that transforms a use of macro to a code formulated entirely in the core language.

### 2.3 Hygienic Macro Systems

A syntactic macro system is called *hygienic* if it guarantees a property known as the *hygienic condition*. It is absence of accidental name collisions between the variables introduced by the macro expander and those declared in the lexical context of the macro use. The hygienic macro system was invented by Kohlbecker and others in 1986 [18] and its expressiveness, implementation techniques, and debugging methods were later studied in [7, 10, 13, 15].

The hygienic macro system was accepted as part of the Scheme standard (R$^4$RS [1]). A Scheme macro is defined by a set of pattern-matching rules. For example, you can define

[1] The number of non-terminal symbols in the syntax definition for Objective Caml 1.11 is 138, while an S-expression is either an atom or a pair.

an or macro as below, which mimics the behavior of one of Scheme's built-in special forms:

```
(define-syntax or
  (syntax-rules ()
    ((or) #f)
    ((or e) e)
    ((or e1 e2 ...)
     (let ((t e1)) (if t t (or e2 ...))))))
```

Because Scheme supports multiple macro systems and also user-defined macro systems, the macro declaration for the or specifies which macro system to be used for this macro. The **syntax-rules** designates the hygienic macro system incorporated in Scheme to be used for the expansion of this macro.

The definition of the or macro consists of three rules. Each of them matches a case where the macro is used with a specific number of arguments. When used with no argument, the or macro expands into a false value. With one argument, it expands into the argument form itself. The last rule is for the case where there are more arguments. In this case, the first macro argument is bound to the *e1* variable and the rest is bound to the "*e2* ..." pattern, which denotes a repetition of one or more forms. Macro expansion is performed by filling the expansion part of the third rule, e.g., "(**let** ((*t e1*)) ...)," with forms bound to macro variables.

The expansion part of the third rule uses the or macro itself, making it a recursively-defined macro. Scheme's macro expander recursively expands such macros. For example, "(or *e1 e2 e3*)" expands into the following form.

```
(let ((t1 e1))
  (if t1 t1
    (let ((t2 e2))
      (if t2 t2
        e3))))
```

Little has been discussed about the hygienic condition so far. The hygienic condition is a property of certain macro expanders that guarantees that "generated identifiers that become binding instances in the completely expanded program must only bind variables that are generated at the same transcription step" [18]. Roughly, this property prevents accidental name collisions between the variables introduced in a macro, and free variables or the variables introduced in other macros. Let us consider a problem caused from a non-hygienic macro expansion of a use of the or macro. In the following example, the or macro occurs in a lexical scope that contains a binding for the *t* variable.

```
(let ((t 1))
  (or #f t))
```

A non-hygienic macro expansion may generate the following form.

```
(let ((t 1))
  (let ((t #f))
    (if t t t)))
```

In this expansion, we see five occurrences of *t*. The first one originates in the **let** binding, found in the outer context of the macro use, and the last occurrence is supposed to refer to the first one. The second occurrence was introduced during expansion of the or macro's third rule and the third and fourth ones refer to it. Note, however, that the third through fifth occurrences of *t* are in fact bound by the second **let** form, which accidentally captures the exterior **let** binding that is supposed to cover the fifth occurrence of *t*.

In this example, to avoid or at least minimize the name collision problem, an experienced macro writer would carefully avoid a typical variable name like *t* and instead use a more cryptic one like *_t_or_12345* and hope that variable bindings for the identical identifier never occurs in the macro's surrounding context. However, there are cases where such heuristics do not fully address problems with non-hygienic macro expansion [7]. In case of LISP, a better solution is to generate a fresh variable by *gensym* function for every variable bindings introduced during macro expansion.

Hygienic macro system, on the other hand, automatically introduces a fresh name for each variable binding found in the macro body. An example of hygienic macro expansion of the form above is as follows:

```
(let ((t 1))
  (let ((t_21_22 #f))
    (if t_21_22 t_21_22 t)))
```

The first implementation of hygienic macro expansion appeared in [18]. Later its time complexity was reduced from a quadratic order of the program size to its linear size [7]. In principle, what a hygienic macro expander does is substitution of variable names introduced in macros with fresh ones. It can be regarded as an automated *gensym* insertion mechanism.

## 3. Porting A Hygienic Macro System

This section briefly describes how a rather complicated hygienic macro system can be ported to a non-LISP programming language.

One obvious strategy for porting a system written in one programming language to another is to translate the existing code into one in the target language. In case of Scheme's hygienic macro systems, they are often written in Scheme. Translation becomes difficult when the core programming language that the macro system of choice is to be ported does not offer features powerful enough for symbolic computation. There is also a more serious problem that arises due to the way hygienic macro systems are implemented. They are often implemented using macro-enhanced Scheme code [14]. For their translation into another language to work, either the language needs to have built-in support of a hygienic

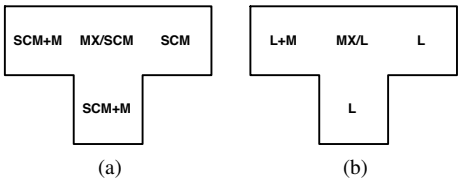


**Figure 1.** Comparison of architectures of psyntax, a reference implementation of R$^6$RS macro transformer (1(a)) and a hygienic macro system for some programming language $L$ (1(b)).

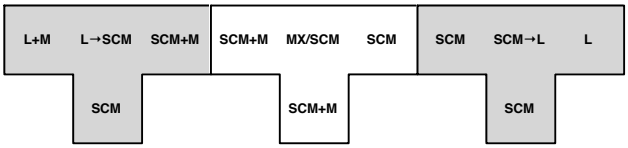


**Figure 2.** We can implement the macro expander for language $L$ by using Scheme's macro expander and making translator to/from Scheme (L→SCM and SCM→L).

macro system, or a version of the portable implementation that is already macro-expanded and thus macro free needs to be chosen.

A reference implementation of Scheme's macro system by Ghuloum and Dybvig consists of about 4,000 lines of code. However, when macro-expanded, it grows into more than 30,000 lines of code. To make the matters worse, most symbols are renamed to cryptic ones like `g$739$12318` to meet the hygienic condition.

Figure 1 illustrates this situation using Bratman's *T-diagram* [5]. The $T$-shape in the diagram represents a program which translates a program written in programming language $L_1$ to a program written in another programming language $L_2$. The translator itself is implemented in programming language $L_0$. Visually, the foot of the $T$-shape is labeled $L_0$ and its left and right arms are labeled $L_1$ and $L_2$, respectively.

The reference implementation of R$^6$RS macro system called *psyntax* [14] is written in macro-enabled Scheme (SCM+M). It takes a Scheme program that may contain definitions and uses of macros (SCM+M) and transforms it to a core Scheme program (SCM). An implementation of a hygienic macro system that we would like to have for language $L$ is depicted in 1(b). It is written in $L$ (L), accepts a program written in macro-enhanced extension of $L$ (L+M) as its input, and macro-expands it into a macro-free program written in $L$ (L).

Instead of porting Scheme's hygienic macro system from scratch, we propose an implementation scheme in which *two-way conversion* between $L$ and Scheme is provided and macro-expansion is handled on the Scheme side. The idea is illustrated in Figure 2. In our proposal, an $L$ to Scheme translator (L→SCM) takes a program written in a macro-enhanced extension of $L$ (L+M) and converts it to an equivalent Scheme program (SCM+M), which contains definitions and uses of macros, now coded in Scheme. The translator converts all the definitions and uses of the macros, plus ordinary program fragments, to the Scheme syntax. Then this Scheme program is macro-expanded by the Scheme's hygienic macro expander (MX/SCM) to produce a macro-free Scheme program (SCM). Finally, the reverse translator (SCM→L) takes the output of the hygienic macro expander and translates it back into the $L$ syntax (L). Because the result of this three-step translation produces a macro-free program written in $L$, it runs on any implementation of $L$.

Two technical hurdles of our proposal are implementation of the parser for the macro-enabled extension of $L$ (i.e., L+M) and that of the two-way translator between $L$ and Scheme. The following two sections address these issues.

## 4. Extensible Parser Architecture

This section presents our macro-definition language for JavaScript and explains how extended syntactic forms introduced by user-defined macros are manipulated by the parser.

### 4.1 Our Macro Language and its Features

We chose the *Good Parts* subset of JavaScript [8] as the input core language for our macro system. The Good Parts JavaScript is a fairly large subset that eliminates some language features that are given ambiguous or ill-founded semantics, such as assignment to undeclared global variable. From now on, we will call the Good Parts JavaScript simply JavaScript.

The macro definition language for JavaScript is similar in flavor to the **define-syntax** construct for Scheme. Unlike Scheme, which is an expression language, JavaScript is an imperative programming language, whose syntactic entities are either a statement or an expression. For this reason, our macro definition language for JavaScript provides two declarations, **define_expression** and **define_statement**. They designate which of the two classes of syntactic entities the macros should expand to.

The definition template of expression macros is as follows.

```
define_expression AnExpressionMacro (⟨keyword⟩, ...) {
  identifier: ⟨identifier name⟩, ...;
  expression: ⟨expression name⟩, ...;
  {
    _(⟨expression parameter⟩, ...) =>
    ⟨expansion⟩;
    ⋮
  }
};
```

This declaration introduces a new syntactic form named AnExpressionMacro into JavaScript. Macro expansion rules of a macro is defined by one or more pattern-matching

rules of the following form: "⟨*pattern*⟩ => ⟨*expansion*⟩." Macro writers can use an arbitrary number of pattern variables in the expansion rules, whose syntactic categories are annotated either by an **identifier:** declaration or an **expression:** declaration. The declaration of syntactic categories is required for generating macro-specific parsing modules. As we will see shortly, when the macro expander accepts a new macro definition, it generates a custom parsing module for the macro and has the parser load it. For direct generation of this custom parsing module, the macro definition needs to specify syntactic categories of macro parameters used in the definition.

Statement macros have a slightly different syntax before it accepts statement parameters as well as identifier and expression parameters.

```
define_statement AStatementMacro (⟨keyword⟩, ...) {
  identifier: ⟨identifier name⟩, ...;
  expression: ⟨expression name⟩, ...;
  statement:  ⟨statement name⟩, ...;
  {
    _(⟨expression parameter⟩, ...) {
      ⟨statement parameter⟩;
      ...
    } => ⟨expansion⟩;
    ⋮
  }
};
```

In addition to identifiers and expressions that are used for expression macros, a statement macro accepts a sequence of statements enclosed in curly braces. To handle such statements, **statement:** declaration is introduced for **define_statement**. For example, **For-in** statement can be defined using the following template.

```
define_statement For (in) {
  identifier: item;
  expression: collection;
  statement: s1, s2;
  { _(item in collection) { s1; s2; ... } =>
    {
      collection.iter(function (item) {
          s1; s2; ...
        });
    }
  }
}
```

The For-in macro thus defined can be used to scan each element in an object that provides an *iter* method.

```
For (person in People) person.do_something();
```

### 4.2 Extensible Parsing Architecture Based on Top-Down Operator Precedence

To process user-defined macros, the macro expander has to incorporate an extensible parser which can be dynamically extended with the ability to parse the additional syntax introduced by the macros.

To add the ability to extend the JavaScript parser with user-defined macros, we adopt *top-down operator precedence* parser by Pratt [20]. In essence, it is a kind of recursive descent parser that incorporates operator precedence. Unlike the more commonly used LALR grammar, which is a bottom-up parsing technique, a top-down operator precedence parser starts parsing from the starting symbol of the grammar and gradually goes down along derivation rules. Top-down operator precedence is often used in handcrafted parsers, while the LALR grammar is more often used for building compiler-compilers. Crockford describes the beauty of this parsing technique in [9]. The implementations of his parser and JSLint are publicly available[2].

One reason we decided to incorporate top down operator precedence is because we wanted to take advantage of the existing and reliable implementation of the JavaScript parser done by Crockford. However, the more important reason was that top-down operator precedence allows us to modularize the parser into a set of independent parsing modules and have the parser load additional parser modules during its execution.

As we already mentioned, a top-down operator precedence parser is a kind of recursive descent parser. The parser consists of a set of functions each of which roughly corresponds to the parsing of a non-terminal symbol used in the grammar. Syntactic ambiguity is controlled by the operator precedence given to all the terminal symbols. For this purpose, Pratt introduced two precedence values called *left-* and *right-binding-power*.

Syntax extensibility can be achieved in the following way. A JavaScript parser is put together as a set of parsing modules. Each parsing module serves as a parser for a non-terminal symbol in the grammar. Parsing is performed by dispatching parsing of a non-terminal symbol to the corresponding parsing module. When the parser reads a macro definition, it generates a parsing module that corresponds to the macro pattern as a first-class function. A macro pattern consists of a sequence of keywords, parameters, and elements, enclosed by parentheses or curly braces. If it is a keyword, it simply generates a function that just reads in the keyword. If it is a macro parameter, it generates a parser function of the corresponding syntactic category of the parameter (e.g., *parse_expression* for an expression parameter). If it is enclosed elements, it generates a function that reads in the block-opening and closing symbols, and put between them a call to the parser that reads in the enclosed elements.

[2] See `http://javascript.crockford.com/tdop/tdop.html` and `http://www.JSLint.com/lint.html`.

### 4.3 Current Implementation: Expression Macro Parser

Our current prototype implementation is a tiny subset of the proposed macro language. It supports definition of expression macros only. Usage of expression macros appears to be exactly the same as function application. When our parser accepts what appears to be a function application form, it checks to see if the function position of the application form is a symbol that denotes a user-defined macro name. If that is the case, it adds a tag that marks that it is a macro use. This tag is used by the macro expander, which will be described below.

## 5. Macro Expander

In the previous section, we saw how a macro-enhanced JavaScript program is parsed. Our macro expander takes a parse tree as its input, converts it to an equivalent Scheme code, has Scheme's macro expander `macro-expand` the Scheme code and generate macro-free Scheme code, and finally converts it back to produce a macro-free JavaScript program.

### 5.1 Conversion to/from Scheme

In the initial parsing phase, a program in extended JavaScript is read in and its abstract syntax tree is produced. In the next phase, macro definitions found in the JavaScript code are translated into a Scheme code that defines equivalent macros. Other declarations and statements that exist in the abstract syntax tree are also converted to S-expressions as outlined in Figure 3.

The literal constants (denoted by $c$) in the code are translated into Scheme equivalents, $c_S$. Null, Boolean, Number, String literals are represented by identical Scheme strings tagged by their type names as follows:

```
(JS const "null")  (JS const "true")
(JS number "0x0012")
(JS number "3.14159265358979323846")
```

Note we are *not* trying to build a JavaScript-to-Scheme compiler. Our ultimate goal is to expand macros in an extended JavaScript code into an equivalent JavaScript code that contains no macros; it is not to run the converted code on a Scheme system. It thus suffices for our purposes that conversion between two programming languages maintains structural isomorphism and some of the semantic equivalence that is related to the process of macro expansion. As for constant literals, we do not translate JavaScript literals into their semantically equivalent Scheme literals; we tag their string representation instead. By doing so, we eliminate possible misinterpretation and approximation errors due to the differences in specifications of the two programming languages and different techniques applied to their implementations. For example, it would cause some problems if a macro expander converted `0x0012` to `18`. For another, it would be a bug for the macro expander to convert a floating point number `3.14159265358979323846` to a less precise number such as `3.141592653589793`.

Tagging constant literals makes converting back to JavaScript from Scheme straightforward. It also guarantees that all the constant literals are converted back to exactly the same ones.

It could become a source of bugs if Unicode characters and Unicode's escape sequences appeared in an identifier's name, but we currently ignore such cases.

Unary and binary operators are represented by **(JS op1 "⊖" ...)** and **(JS op2 "⊕" ...)** forms where ⊖ and ⊕ stand for a unary symbol and a binary one, respectively.

Initializers and accessors of JavaScript objects, and arrays are also represented by tagged S-expressions. Because the language specification of Scheme does not define an object system, converting the *semantics* of JavaScript's object system into Scheme would require a large amount of effort. Here again, we simply convert the program structure of the JavaScript to an equivalent S-expression using **JS object** and **JS field** (pseudo) special forms.

As for vectors, since Scheme does provide vectors similar to those in JavaScript, it would not be difficult to convert JavaScript-style array initializers and accessors to corresponding Scheme forms. However, conversion would be more easily implemented by incorporating pseudo special forms such as **JS array** and **JS index**. Also, JavaScript and Scheme behave differently in the case of out-of-bound access to vectors: JavaScript returns an **undefined** value, but Scheme throws a runtime error. As this example illustrates, semantically correct translation of a language to another in the strict sense is a hard problem and should be avoided whenever it can be.

Control structures such as **if** and **for** statements are converted to **JS if** and **JS for** pseudo special forms, respectively. Because **for** is also a syntactic form in Scheme, which often is defined by a macro, if we do not protect occurrence of JavaScript's **for** stetement by the **JS** tag, it can be expanded to lower-level syntactic forms on the Scheme side. Our **JS**-tagging technique effectively avoids such chance.

### 5.2 Conversion of Variable Bindings

Now, in this conversion, just what should we transfer from a JavaScript program to its equivalent Scheme program? So far, we just convert the structure of the JavaScript program using pseudo special forms and it seems that we are not interested to converting the semantics of the JavaScript program to Scheme. One exception is that the conversion preserves JavaScript variable names to equivalent Scheme variable names. For instance, a variable $v$ is converted to itself and a JavaScript-style variable declaration "**var** $v$ `=` ..." is converted to a Scheme-style variable declaration, i.e., "**define** $v$ ...". Semantically correct treatment of variables and their bindings is at the heart of hygienic macro system. It is therefore important for the conversion process to preserve the

$$T[c] = c_S \quad T[v] = v \quad T[\ominus e] = (\textbf{JS op1 "}\ominus\textbf{"}\ T[e]) \quad T[e_1 \oplus e_2] = (\textbf{JS op2 "}\oplus\textbf{"}\ T[e_1]\ T[e_2])$$

$$T[\textbf{var}\ v\ =\ e] = (\textbf{define}\ v\ T[e])$$

$$T[\{\ \ell_1 :\ v_1,\ \ell_2 :\ v_2,\ \ldots\}] = (\textbf{JS object}\ T[\ell_1]\ T[v_1]\ T[\ell_2]\ T[v_2]\ \ldots) \quad T[o.\ell] = (\textbf{JS field}\ T[o]\ T[\ell])$$

$$T[[\ e_1,\ e_2,\ \ldots\ ]] = (\textbf{JS array}\ T[e_1]\ T[e_2]\ \ldots) \quad T[e_1[e_2]] = (\textbf{JS index}\ T[e_1]\ T[e_2])$$

$$T[\textbf{if}\ (e)\ \{\ E_1\ \}\ \textbf{else}\ \{\ E_2\ \}] = (\textbf{JS if}\ T[e]\ T[E_1]\ T[E_2]) \quad T[\textbf{for}\ (e_1; e_2; e_3)\ \{\ E\ \}] = (\textbf{JS for}\ T[e_1]\ T[e_2]\ T[e_3]\ T[E])$$

$$T[\textbf{function}\ f(v_1, v_2, \ldots)\ \{E\}] = (\textbf{define}\ (f\ v_1\ v_2\ \ldots)\ T[E]) \quad T[\textbf{function}\ (v_1, v_2, \ldots)\ \{E\}] = (\textbf{lambda}\ (v_1\ v_2\ \ldots)\ T[E])$$

$$T[f(e_1, e_2, \ldots)] = (\textbf{JS application}\ T[f]\ T[e_1]\ T[e_2]\ \ldots)^{\dagger} \quad T[\textbf{return}\ e] = (\textbf{JS return}\ T[e])$$

$$T[\textbf{define\_syntax}\ \mathsf{macro}\ ()\ \ldots\ \{\ \textit{pattern}\ \texttt{=>}\ \textit{expansion};\ \ldots\ \}] = (\textbf{define-syntax}\ \mathsf{macro}\ (\textbf{syntax-rules}\ ()\ (\textit{pattern}\ \textit{expansion})\ \ldots))$$

$$T[f(e_1, e_2, \ldots)] = (f\ T[e_1]\ T[e_2]\ \ldots)^{\ddagger}$$

**Figure 3.** Conversion of JavaScript code fragments to Scheme. Conversion function $T$ takes a JavaScript code fragment, $E_{\text{JS}}$, and converts it to an equivalent Scheme form, $E_{\text{S}}$. In other words, $T[E_{\text{JS}}] = E_{\text{S}}$. Both function and macro application forms have identical structures ($f(e_1, e_2, \ldots)$). When such an application form is found, the converter checks to see if the element at the function position (i.e., $f$) is a name and is also already defined as a macro. If it is the case, that form is treated as a macro application (i.e., ‡-rule); otherwise, as a function function application (i.e., †-rule).

variable binding structure in the original JavaScript program when it is converted to Scheme.

For this purpose, syntactic elements that create variable bindings should be carefully translated into equivalent Scheme forms. Figure 3 includes conversion rules for two styles of function declarations, both of which create bindings for the formal function parameters, namely $v_1, v_2, \ldots$.

Both function and macro application forms have a seemingly identical structure: "$f(e_1,\ e_2,\ \ldots)$." When such a form is identified, the converter checks to see if the element at the function position (i.e., $f$) is an identifier. If it is, then it checks if it is a macro name. If so, the form is treated as a macro application (i.e., ‡-rule in Figure 3); otherwise, as a function function application (i.e., †-rule). Finally return statements are represented by respective pseudo special forms.

### 5.3 An Example of the Conversion

As an example of this conversion process, let us consider the following code that defines and uses a Let macro.

```
define_expression Let() {
  identifier: id;
  expression: expr, body;
  {
    Let(id, expr, body) =>
    ((function (id) {
        return body;
     })(expr));
  }
};

var info =
  Let(id, uname + (++serial),
      { email: id + mail_domain,
        web: web_domain + id });
```

In converting the JavaScript code, the converter splits the code into two Scheme code fragments. One is a collection of Scheme-style macro definitions and the other is the rest of the code.

The definition for the Let macro is translated into the following Scheme-style definition.

```
(define-syntax Let
  (syntax-rules ()
    ((Let id expr body)
     (JS application
       (lambda (id) (begin (JS return body)))
       expr))))
```

This macro definition may look like a typical macro definition in Scheme but it contains our pseudo special forms such as **JS application** and **JS return** which standard Scheme does not understand. Although Scheme macro expander accepts this macro definition and can expand the uses of the Let macro, the expanded pseudo Scheme form does not run on a Scheme system; the macro-expanded S-expressions are meant to be transformed back to JavaScript code.

The rest of the original JavaScript code is converted to the following Scheme-like S-expression.

```
(begin
  (define info
    (Let
      id
      (JS op2 "+" uname (JS op1 "++" serial))
      (JS object
        ("email" (JS op2 "+" id mail_domain))
```

```
        ("web" (JS op2 "+" web_domain id))))))
```

Then the macro uses found in the converted code is expanded using the macro expander of Scheme. We thus obtain the following Scheme code. At this point, the macro definitions are removed, the macro uses are expanded, and the code is free of macros.

```
(begin
  (define _L12
    (lambda (id_27_)
      (JS return
        (JS object
          ("email" (JS op2 "+" id_27_mail_domain))
          ("web" (JS op2 "+" web_domain id_27_))))))
  (define info
    (JS application _L12
      (JS op2 "+" uname (JS op1 "++" serial)))))
```

We can see that the variable *id* introduced in the definition of Let macro is renamed to *id_27_* in the expanded code. Variable name renaming is the most important feature of Scheme's hygienic macro system, which properly renames variable names introduced by macro definitions to avoid name collision against variable names occurring in a macro usage.

Finally, the Scheme code that we just obtained is converted back to JavaScript. This process simply reverses the conversion depicted in Figure 3. The following code, a macro-expanded JavaScript code, is the result for the whole macro expansion process:

```
{
  var _L12 = function (id_27_) {
    return { email: (id_27_ + mail_domain),
             web: (web_domain + id_27_) };
  };
  var info = _L12((uname + (++serial)));
}
```

The parser is implemented by 1,500 lines of Scheme code; and the mutual translator between JavaScript and Scheme in 500 lines of Scheme code. Both are implemented by Gauche Scheme[3] but macro expansion is performed on ypsilon Scheme[4].

## 6. Playing with Macros

```
define_expression or() {
  identifier: tmp;
  expression: exp1, exp2;
  {
    or() => false;
    or(exp1) => exp1;
    or(exp1, exp2, ...) =>
      Let(tmp, exp1, tmp ? tmp : or(exp2, ...));
  }
};
```

The first example is an or macro. This seemingly simple and not-so-useful macro (after all, JavaScript has a built-in logical disjunction operator ||) turns out to be highly suggestive. Firstly, it shows that a feature in a language that can only come built in, normally, can be defined as a user-defined macro.

Secondly, the definition consists of multiple pattern matching rules. The first rule corresponds to the macro usage where it takes no argument; the second, to the one where it takes a single argument; and the last one, to the where it takes more. The last rule uses an ellipsis (...) that denotes repetitive occurrences of the preceding expression, which in this case is an expression macro variable named *exp2*.

Thirdly, the user-defined macro name (i.e., or) happens to collide with one of Scheme's built-in special form: Scheme offers a multi-argument logical disjunctive special form named or. You might fear that loading this user-defined or macro may cause a name collision error or confuse Scheme's macro expander, but that is not the case. Consider the following use of the or macro.

```
var tmp = 10;
or(false, tmp);
```

It is expanded to the following JavaScript code, without any errors. It should also be noted that the hygienic condition holds for the *tmp* variable.

```
{
  var _L12 = function (tmp_26_28_) {
    return ((tmp_26_28_) ? (tmp_26_28_) : (tmp));
  }
  var tmp = 10;
  _L12(false);
}
```

Finally, the or macro is recursively defined; the or macro is used inside its own definition. The expanded form above shows that the recursive expansion is done as expected.

Here is another example for cloning an object. This new macro creates a new object by using an existing object as prototype and extending it with some additional fields.

```
define_expression new() {
  identifier: proto, specs;
  {
    new(proto, specs) =>
    ((function () {
      var new_obj = { __proto__: proto };
      var id;
```

[3] http://practical-scheme.net/gauche/

[4] http://code.google.com/p/ypsilon/. We started our prototype implementation on Gauche and later found out that Gauche does not let us access the macro-expanded forms. After this discovery, we decided to call ypsilon's macro-expander via inter process communication.

```
      for (id in specs) new_obj[id] = specs[id];
      return new_obj;
    })());
  }
};
```

Using this macro, we can create a new object in the following way:

```
var rectangle = { height: 10, width: 20 };
var red_rectangle = new(rectangle, { color: "red" });
red_rectangle.height; // => 10
red_rectangle.color;  // => "red"
```

## 7. Related Work

As we reviewed in Section 2, syntactic macro systems have been implemented mainly for the LISP family. However, there also have been a few attempts to add support for syntactic macros to descendants of ALGOL.

MS$^2$ is a syntactic macro system for the C programming language [25]. It is similar to LISP's **defmacro** in its scope. It offers an API called *explicit code template operators* for programming macro transformers. The API includes quasi-quotation and *gensym* function. Its LISP-style syntactic macro system is flexible but pattern driven; support for hygienic macros is missing. Another weakness with MS$^2$ is that a macro has to be formed as a sequence and it does not support structured macro forms.

Dylan [22] is an object-oriented programming language that supports multimethods. Although Dylan is heavily influenced by the first-classing philosophy of Scheme and Common Lisp's object system, its syntax is in the style of ALGOL. Standard Dylan supports a rewrite-based macro system. Later [3] presented an implementation scheme for a hygienic macro system for Dylan. They claim it is the first implementation of a hygienic macro system for a descendant of ALGOL. Their implementation is based on *D-expressions* and requires heavy modification of the Dylan compiler. Our proposal, in contrast, is light-weight in that its macro expansion relies on Scheme's macro system. Our current implementation consists of about 2,000 lines of Scheme code.

MetaBorg [6], we have already mentioned it in Subsection 2.2, is a meta-programming framework with which programmers can extend the syntax of a programming language by describing the original grammar and user-defined subgrammars. At the heart of their proposal is the *Stratego/XT* compiler infrastructure, which is used to describe the grammars of the host and extended languages. It enjoys the full expressiveness of meta-programming, but macro writers are required to learn how to use the complicated Stratego/XT infrastructure. In our proposal, macro writers only need to learn how to use a simple rewrite-based description framework.

## 8. Conclusions

We proposed a universal implementation scheme of hygienic, syntactic macro systems, using JavaScript as the target language. We first identified two difficulties in its implementation: (1) syntactic extensibility of the parser to deal with user-defined syntactic macros, and (2) logical complexity of the hygienic, syntactic macro system. An extensible parsing architecture based on modularized top-down operator precedence was proposed to address the first problem. To address the second problem, we proposed piggybacking on the hygienic macro system for Scheme, using supplementary JavaScript-to-Scheme and Scheme-to-JavaScript converters. These two strategies, when combined, brought us a hygienic macro system for JavaScript by only a small amount of code. The whole the system was implemented by 2,000 lines of Scheme code.

The current implementation supports expressions only. We plan on adding support statements soon to make our system complete.

## Acknowledgments

We are grateful for the useful comments provided by the anonymous reviewers about an earlier draft of this paper and fruitful discussion that we exchanged during the S3 workshop. We thank our colleagues for their encouragements and constructive criticisms on our work. Also we appreciate Ms. Kanako Homizu for reading an earlier draft of this paper and providing a feedback. Dr. Yasuro Kawata attentively helped us proofread the paper.

## References

[1] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams, IV, D. P. Friedman, E. Kohlbecker, G. L. Steele, Jr., D. H. Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised$^4$ report on the algorithmic language Scheme. *SIGPLAN Lisp Pointers*, IV(3):1–55, 1991.

[2] N. I. Adams, IV, D. H. Bartley, G. Brooks, R. K. Dybvig, D. P. Friedman, R. Halstead, C. Hanson, C. T. Haynes, E. Kohlbecker, D. Oxley, K. M. Pitman, G. J. Rozas, G. L. Steele, Jr., G. J. Sussman, M. Wand, and H. Abelson. Revised$^5$ report on the algorithmic language Scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.

[3] J. Bachrach and K. Playford. D-Expressions: Lisp power, Dylan style. Nov. 1999.

[4] C. Brabrand and M. I. Schwartzbach. Growing languages with metamorphic syntax macros. In *PEPM '02: Proceedings of the 2002 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 31–40, New York, NY, USA, 2002. ACM.

[5] H. Bratman. An alternate form of the "UNCOL diagram". *Communications of the ACM*, 4(3):142, 1961.

[6] M. Bravenboer and E. Visser. Concrete syntax for objects: Domain-specific language embedding and assimilation with-

out restrictions. In *OOPSLA '04: Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383, New York, NY, USA, 2004. ACM.

[7] 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.

[8] D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, Inc., 2008.

[9] D. Crockford. Top down operator precedence. In A. Oram and G. Wilson, editors, *Beautiful Code: Leading Programmers Explain How They Think*, chapter 9, pages 129–145. O'Reilly and Associates, June 2007.

[10] R. Culpepper and M. Felleisen. Debugging hygienic macros. *Science of Computer Programming*, 75(7):496–515, 2010.

[11] D. de Rauglaudre. Camlp4 - reference manual. `http://caml.inria.fr/pub/docs/manual-camlp4/index.html`, Sept. 2003.

[12] R. K. Dybvig. Writing hygienic macros in Scheme with syntax-case. Technical report, Indiana Computer Science Department, 1992.

[13] R. K. Dybvig. Syntactic extension. In *The Scheme Programming Language, 4th Edition*, chapter 8. The MIT Press, 2009.

[14] A. Ghuloum and R. K. Dybvig. R$^6$RS libraries and syntax-case system. http://ikarus-scheme.org/r6rs-libraries/.

[15] 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*, pages 48–62, Berlin, Heidelberg, 2008. Springer-Verlag.

[16] B. W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 1988.

[17] D. E. Knuth. *The TeXbook*. Addison-Wesley Professional, 1986.

[18] 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.

[19] B. M. Leavenworth. Syntax macros and extended translation. *Communications of the ACM*, 9(11):790–793, 1966.

[20] V. R. Pratt. Top-down operator precedence. In *POPL '73: Proceedings of the First Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 41–51, New York, NY, USA, 1973. ACM.

[21] R. Seindal, F. Pinard, G. V. Vaughan, and E. Blake. GNU m4 manual. `http://www.gnu.org/software/m4/manual/m4.pdf`, Mar. 2009.

[22] A. Shalit. *The Dylan Reference Manual: The Definitive Guide to The New Object-Oriented Dynamic Language*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[23] G. L. Steele, Jr. *Common Lisp: the Language*. Digital Press, Newton, MA, USA, second edition, 1990.

[24] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Professional, 2002.

[25] 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.