情報処理学会論文誌　プログラミング　Vol.6　No.2　85–101 (Aug. 2013)

解析表現文法とSchemeマクロ展開器を用いたJavaScript向けHygienic構文マクロシステムの実装

甫水 佳奈子[1,†1,a)]　脇田 建[1,b)]　佐々木 晃[2,c)]

受付日 2012年11月12日, 採録日 2013年4月24日

概要：本稿は，JavaScript の構文拡張を可能にする Hygienic 構文マクロシステムの実装技法を提案する．Hygienic 構文マクロシステムは，マクロ展開の前後で変数の束縛や参照関係を破壊しない安全な構文マクロシステムである．この Hygienic 構文マクロシステムを利用することによって，プログラミング言語の構文の自由な拡張が可能になる．しかし，Hygienic 構文マクロシステムは，S 式という一貫した構文構造を持つ Scheme には標準で組み込まれているものの，その他の一般的なプログラミング言語に実装された例はほとんどない．本稿では，まず，汎用的なプログラミング言語における Hygienic 構文マクロシステムの実装の難しさを示し，次に，本研究が提案する JavaScript 向け Hygienic 構文マクロシステムの実装技法について述べる．提案する実装技法では，マクロ構文の追加によって拡張される JavaScript 構文を解析するための拡張可能なパーザの実現に解析表現文法を用い，マクロ展開は既存の Scheme マクロ展開器に委ねる．マクロ展開においては，マクロを含む JavaScript コードをそれと等価な S 式へと変換し，Scheme マクロ展開器で展開を行った後に，JavaScript コードに逆変換するという言語間相互変換を行う．これらの工夫によりわずか 2,000 行弱のコンパクトな実装によって JavaScript に対する記述力が高い Hygienic 構文マクロシステムを実現できた．

キーワード：Hygienic 構文マクロシステム，解析表現文法，JavaScript，Scheme

An Implementation of a Hygienic Syntactic Macro System for JavaScript Using Parsing Expression Grammar and a Scheme Macro Expander

Kanako Homizu[1,†1,a)]　Ken Wakita[1,b)]　Akira Sasaki[2,c)]

Received: November 12, 2012, Accepted: April 24, 2013

Abstract: The article introduces an implementation scheme for a hygienic syntactic macro system that adds syntactic extensibility to JavaScript. A hygienic syntactic macro system guarantees certain level of semantic soundness by ensuring variable binding and referencing semantics to be maintained before and after macro expansion. Although a hygienic macro system is a powerful tool for extending programming language syntax, its implementation is found only in Scheme and few other programming languages. The article identifies technical difficulty for adding a hygienic macro feature to programming languages other than LISP and then describes a simple implementation scheme, where the extensible parser is built by dynamic generation of parsing expression grammar descriptions, and macro expansion is performed by delegating the task to Scheme. Our macro expander firstly translates the macro-enhanced JavaScript code to equivalent S-expression, lets the macro expander of Scheme expand the S-expression, and finally translates the macro-free S-expression back to macro-free JavaScript source program. This smart implementation scheme allows us to implement an expressive hygienic macro system with less than 2,000 lines of code.

Keywords: Hygienic syntactic macro system, parsing expression grammar, JavaScript, Scheme

[1] 東京工業大学大学院情報理工学研究科数理・計算科学専攻
Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, Meguro, Tokyo 152–8552, Japan
[2] 法政大学情報科学部
Faculty of Computer and Information Sciences, Hosei University, Koganei, Tokyo 184–8554, Japan

[†1] 現在，三菱電機株式会社
Presently with Mitsubishi Electric Corporation
[a)] homizu.k@gmail.com
[b)] wakita@is.titech.ac.jp
[c)] asasaki@hosei.ac.jp

1. はじめに

プログラミング言語の構文拡張を可能にするマクロシステムは，プログラムの簡略化やドメイン特化型言語の作成に大いに役立つ機構である．なかでも，*Hygienic* 構文マクロシステムは，プログラム中の変数の束縛関係を壊さずに構文木の置き換えを行う安全なマクロシステムである．しかし，それは Lisp 言語の一種で S 式という一貫した構文構造を持つ Scheme [1] には標準で実装されているものの，その他の一般的なプログラミング言語にはほとんど実装されていない．

本稿では，非 Lisp 言語における Hygienic 構文マクロシステムの実現に向けて，JavaScript [15] 向けの Hygienic 構文マクロシステムについて述べ，そのための実装上の問題点を指摘したうえで，系統的な実装手法を提案する．既存の構文マクロシステムは抽象構文木の構造をマクロプログラマが直接操作するための低レベルの API を提供する方式がほとんどであるのに対して，本提案はホスト言語の構文に沿ったパターンマッチによってマクロを定義できる．このため，マクロプログラマの負担が大幅に軽減される．

パターンマッチによって構文マクロを定義する方式は Scheme の Hygienic マクロシステムにはじめて導入された．本研究でもその設計を踏襲したが，括弧を多用する Scheme の文法に対して非 Lisp 系の端正な構文をユーザ定義マクロにおいても実現できるように，マクロプログラマがキーワードや句読点，構文要素のグループを柔軟に指定できる仕組みを導入した．この拡張によって，従来のマクロシステムでの括弧を多用する入れ子構造を排除できるようになった．

ここまで述べたように，本稿で提案するマクロシステムは，パターンマッチを用いた構文マクロシステムであり，従来のマクロシステムよりも柔軟なマクロ構文の定義を許し，さらにマクロ展開器は変数束縛の意味論を保持する *Hygienic* 性を有する．このようなマクロシステムの実装上の問題点は，主に (1) 拡張可能な構文解析器を実装すること，そして (2) Hygienic なマクロ展開器を提供することの 2 点である．

ユーザが定義したマクロ構文を構文解析するためには，ホスト言語の構文解析器を拡張し，ユーザ定義マクロで導入される新たな構文に対応することが求められる．このため，拡張可能な構文解析手法が必要になる．これに対する Arai らのアプローチは再帰下降式演算子順位構文解析に基づいた，やや場当たり的な方式であった [2]．本稿では，ユーザ定義マクロを解析表現文法に変形する系統的な方式を提案する．

Scheme の Hygienic なマクロ展開器の実装技術の成熟は，初期の提案からプログラムの大きさについて線形の計算量で可能となる [5] までに多大な努力を要した．本研究では，複雑な Hygienic マクロシステムの実装をすることをあえて避け，JavaScript と Scheme の S 式間で相互変換し，Scheme 側でマクロ展開する方法を提案している．この方式をとることによって実装は簡素化され，さらに実装の大部分はホスト言語との独立性を保つことができた．このことは，本研究で実装した JavaScript のマクロシステムを今後，他のプログラミング言語に容易に転用できる可能性を示唆している．

以下に本稿の構成を述べる．2 章で Scheme の Hygienic 構文マクロシステムについて解説し，3 章で本研究が提案する JavaScript 向け Hygienic 構文マクロシステムの主な仕様について述べる．そして，4 章で非 Lisp 言語における Hygienic 構文マクロの実装の難しさを示した後，5 章で本研究が提案する実装手法について述べる．6 章では，本システムの記述力を支えるその他の機能を紹介し，7 章では実装方法の評価を行う．さらに，8 章で本システムが与える制約や記述力について議論し，9 章で関連研究を紹介する．そして最後に 10 章で本稿をまとめ今後の課題について述べる．

2. Hygienic 構文マクロシステム

マクロシステムはプログラミング言語に拡張性を与える機構として，広く利用されてきた．一般に，マクロの定義は，マクロ構文がとりうる入力形式を表すパターンとその置き換え後の形式を表すテンプレートからなる．そして，このパターンとテンプレートの組によって定義された置き換え規則にしたがい，マクロ展開が行われる．

マクロシステムには，その置き換え対象により字句マクロと構文マクロがある．

CPP [12] や m4 [17]，TEX [13] などのマクロで知られる字句マクロは，その名のとおり字句の置き換えを行うマクロシステムである．マクロ定義によって定義された置き換え規則に従い，字面上の置き換えを行う．ここで，字句マクロはプログラムの構文構造を考慮しないマクロ展開を行うため，プログラマが意図しない展開結果を生む恐れがある．そのため，プログラマはマクロ定義時，使用時ともにマクロがプログラムの構文構造を破壊しないよう細心の注意を払う必要がある．

一方，Lisp 言語の伝統的なマクロとして知られる構文マクロは構文木の置き換えを行うマクロシステムであるため，字句マクロのような問題はない．さらに，構文木を扱うことから，より複雑な構文の定義も容易である．たとえば，CommonLisp [22] のオブジェクトシステムにあたる CLOS はマクロを用いて実装されたものである．ただし，単純な構文マクロには，マクロ展開によって導入された変数が他の変数と衝突してしまう変数捕捉の可能性がある．そのため，プログラマは自らが変数名に衝突しにくい複雑な名前を付けるなどして対処しなければならない．

このような変数捕捉を解決するために，Kohlbecker ら [14] によって提案された概念が Hygienic 性である．Hygienic 性とは，マクロ展開によって導入された変数と，自由変数もしくは他のマクロ展開によって導入された変数との衝突を生じない性質である．そして，この性質を備えた構文マクロシステムが Hygeinic 構文マクロシステムである．この Hygienic 構文マクロシステムは Scheme [1] の標準機能として提供されている．

Scheme のマクロの例として，**syntax-rules** を用いた **or** マクロの定義を**図 1** に示す．この 2〜6 行がマクロの置き換え規則を定義する部分である．**syntax-rules** マクロにおいて，置き換え規則は（⟨*pattern*⟩ ⟨*template*⟩）の形式で記述する．**or** マクロには，引数が 0 個のとき（3 行目）と 1 個のとき（4 行目），1 個以上のとき（5 行目）の 3 種類のパターンがあり，それぞれについて置き換え規則が定義されている．

ここで，**or** マクロの定義の 6 行目で t という変数を用いているが，あえてこの変数と同じ変数 t をマクロの外側で用いた次の式をマクロ展開することを考えてみよう．

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

これを非 Hygienic なマクロ展開器で展開すると，**図 2**（左）のように元々あった 1 行目の t の変数束縛とマクロ展開によって導入された 2 行目の t の変数束縛が衝突してしまい，本来 1 行目の t を参照するはずだった 3 行目の最後の t が 2 行目のものを参照してしまう．そのため，展開後の式を評価すると，本来の 1 ではなく `#f` が結果として得られる．

一方，Hygienic マクロ展開の場合，図 2（右）のように束縛変数はすべて名前が付け替えられる．つまり，α 変換が施される．これにより，マクロ展開によって導入された変数や変数束縛が他のものと衝突する恐れはない．

以上のように，Hygienic 構文マクロシステムはプログラムの構文構造と変数束縛に関する意味構造を破壊しない安全なマクロシステムである．しかしながら，その有用性とは裏腹に，それを Scheme 以外の言語に実装した例はほとんどない．

```
1 (define-syntax or
2   (syntax-rules ()
3     ((or) #f)
4     ((or e) e)
5     ((or e1 e2 ...)
6      (let ((t e1)) (if t t (or e2 ...))))))
```

図 1　**syntax-rules** を用いた **or** マクロの定義
Fig. 1　A definition of **or** macro using **syntax-rules**.

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

```
1 (let ((t1 1))
2   (let ((t2 #f))
3     (if t2 t2 t1)))
```

図 2　(**let** ((*t* 1)) (**or** #**f** *t*)) を非 Hygeinic マクロ展開した結果（左）と Hygienic マクロ展開した結果（右）
Fig. 2　A result of a non-hygienic macro expansion of (**let** ((*t* 1)) (**or** #**f** *t*)) (left fig.) and a result of a hygienic macro expansion of it (right fig.).

3.　提案マクロシステム EX-JS

非 Lisp 言語における Hygienic 構文マクロの実現に向けて，本研究では JavaScript 向け Hygienic 構文マクロシステム EX-JS を提案する．本システムは，Scheme の **syntax-rules** マクロに強く影響を受けたパターンマッチ形式のマクロである．ユーザがソースコードレベルでの置き換え規則を定義するだけで，構文の拡張が可能となる．本マクロシステムを用いて拡張できる構文は式と文であり，式に相当するマクロ構文を**式マクロ**，文に相当するマクロ構文を**文マクロ**と呼ぶ．式マクロと文マクロはそれぞれ，**expression** 宣言子と **statement** 宣言子を用いて定義する．

図 3 に文マクロの例として **unless** マクロの定義と使用を示す．**unless** マクロは，与えられた条件が満たされない場合にのみ文を実行するための構文である．

この最初の 5 行が **unless** マクロの定義である．まず，1 行目で **unless** というマクロ名を定義している．次に，2〜3 行目でマクロ定義の置き換え規則の中でマクロの入力要素を参照するためのマクロ変数の宣言を行っている．JavaScript は Scheme とは異なり，1 つの構文はさまざまな構文要素から構成される．そのため，これから定義するマクロがどのような構文要素から構成されているかを示さなければならない．それを，このマクロ変数の宣言によって行う．各宣言子は，マクロ変数が JavaScript のどの構文要素に該当するかを表しており，この例では，式に相当する **expression**，文に相当する **statement** 宣言子が用いられている．このほかにも識別子に相当する **identifier** 宣言子などがある．4 行目はマクロの置き換え規則を定義している．置き換え規則は，{ ⟨*pattern*⟩ => ⟨*template*⟩ } の形式で記述する．パターンにはマクロ構文の入力形式を記述

```
1  statement unless {
2    expression: C;
3    statement: S;
4    { unless (C) S => if (!C) S }
5  }
6  var c1 = false;
7  function test(c1, c2) {
8    if (c1)
9      unless (c2) console.log('P1');
10   else console.log('P2');
11 }
12 test(true, true);  // 出力なし
13 test(true, false); // P1を出力
```

図 3　**unless** マクロの定義と使用
Fig. 3　A definition of **unless** macro and its usage.

```
⟨pattern⟩ ::= ⟨JS_Identifier⟩ ⟨ps⟩
⟨ps⟩      ::= ⟨p⟩ {⟨p⟩} [...] {⟨p⟩}
⟨p⟩       ::= ⟨JS_Literal⟩
            | ⟨identifier_variable⟩
            | ⟨expression_variable⟩
            | ⟨statement_variable⟩
            | ⟨keyword⟩
            | ( ⟨ps⟩ ) | { ⟨ps⟩ } | [ ⟨ps⟩ ] | [# ⟨ps⟩ #]
```

図 4　パターンの拡張 BNF

Fig. 4　An extended BNF for a pattern.

し，対応するテンプレートはその置き換え後の文（式マクロの場合は式）を記述する．パターンの構文を**図 4** に示す．パターンは JavaScript のリテラル，マクロ変数，キーワードである**単純マクロパターン**と，部分パターンの列，そして部分パターンを括弧記号によりグループ化したものである**複合マクロパターン**から構成される．本システムでは，マクロの使用をマクロ名で判別するため，Scheme の仕様書 R6RS [21] のマクロの仕様と同様にパターンは必ずマクロ名から始まるものとしている．つまり，4 行目で **unless** の代わりに別の識別子を書いても，それは無視され，**unless** に置き換えられる．このため，本システムでは，マクロ名で始まらない構文や新しい演算子の定義などはできない．テンプレートには，式マクロ・文マクロの種別に応じて，JavaScript の式・文を記述する．ここまでがマクロ定義である．

この例の 9 行目では **unless** マクロを使用している．本稿ではパターンにマッチするマクロの使用を**マクロ使用**と呼ぶこととする．このマクロは文マクロであるため，JavaScript の文が出現できる箇所で利用できる．

図 3 をマクロ展開して得られたコードを**図 5** に示す．この展開例には構文マクロの特徴が現れている．字句マクロの場合，この例のように **if** 文の文脈の中に **else** 節を持たない **if** 文に展開されるマクロを使用すると，本来，**if** 文の文脈にあった **else** 節が，マクロ展開に由来する **if** 文に結合され，プログラムの構造が変化する事故が起きうる．一方，この展開例では，マクロ使用に由来する **if** 文は，元々の **if** 文脈の **then** 節に閉じ込められ，元々の **else** 節が結合することが避けられている．また，Hygienic マクロ展開の特徴として，1 行目のグローバル変数 *c1* の名前はそのままであるが，2〜4 行目に出現する関数の仮引数はすべて *c1_199_*，*c2_199_* のように名前が変更されている．このように，関数によってつくり出されたローカルスコープにおける束縛変数は確実に名前が付け替えられるので，本システムのマクロ展開において変数衝突が起こる危険性はない．

次に，式マクロの例として**図 6** に **let** マクロを示す．これは，*body* の評価時に変数 v に e の評価値を束縛するためのマクロである．

この例では，4 行目で **keyword** 宣言子によって，キーワードの定義がなされている．これは，**=** と **In** が **let** の

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

図 5　図 3 をマクロ展開した結果

Fig. 5　A result of a macro expansion of Fig. 3.

```
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
```

図 6　**let** マクロの定義と使用

Fig. 6　A definition of **let** macro and its usage.

中でキーワードとして扱われ，マクロ使用において文字どおりの入力が必要になることを表す．また，この例では省略記号 `...` が用いられているが，これは R6RS のマクロと同じように直前のパターンの 0 個以上の繰返しを表すものである．

さて，この例で特筆すべきは，`[# #]` である．これは，その後にある `...` の繰り返し範囲を明示するための仮想的な括弧記号で，この例では `[# #]` に囲われた $v = e$ という部分パターンが繰り返されることを意味している．`[# #]` は仮想的な記号であるため，実際のマクロ使用に入力として記述してはならない．パターンをグループ化するためのメタ記号には図 3 の例で用いた `( )` のほかにも `{ }`，`[ ]` があるが，`[# #]` を導入することで，括弧記号を用いないより自然な構文の記述が可能となった．

これまではパターンを 1 つしか持たないマクロの例を紹介したが，複数のパターンを持たせることも可能である．たとえば，**図 7** の **Append** マクロは，JavaScript API の jQuery を用いて HTML の DOM 要素にコンテンツを追加するためのマクロで，コンテンツが 1 つのときとそれ以上のときとで 2 つのパターンを持つ．この例のように，マクロを再帰的に定義することもできる．

ここで，マクロを定義するためには，そのマクロのテンプレート部の構文解析が必要になることから，テンプレートに出現するマクロの入力の構文解析について考える．一般にマクロの構文解析は各パターンが示すマクロ使用の構

```
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 = Append Title, Menu, Content
9                to jQuery('body');
```

図 7　**Append** マクロの定義と使用
Fig. 7　A definition of **Append** macro and its usage.

文に沿って行われる．しかし，テンプレートに現れるマクロの入力には，省略記号 ... のようなマクロ使用には現れない構文が含まれるため，単純にパターンが示す構文に沿って構文解析を行うと失敗する．そのため，マクロがテンプレートに出現する場合の構文も想定しておかなければならない．

例として，図 7 の 6 行目のマクロの入力（下線部）を構文解析することを考える．まずパーザは先に定義された 4 行目のパターンに対応するマクロ使用として下線部の構文解析を試みる．しかし，マクロ使用の構文には含まれないコンマが入力に出現するため失敗する．次に，5 行目のパターンに対応するマクロ使用として構文解析を行うが，5 行目の *c2, ...* という部分パターンに対応するマクロ使用の入力にはコンマ区切りの 0 個以上の式を想定しているため，対応する位置に ... が出現する下線部の入力は構文解析に失敗する．以上の考察より，テンプレートに出現するマクロの入力に対して，パターン中で省略記号があった位置に ... の出現を許すような構文が求められることが明らかとなった．これにより，5 行目の *c2, ...* に対応する入力として，テンプレート内に限り，コンマ区切りの 1 個以上の式の後に省略記号 ... が置かれる形式も記述可能とした．しかし，この単純な方策だけでは十分ではなく，構文解析が失敗してしまうことがある．すなわち，5 行目の *c2, ...* に対応する入力には下線部の ... のみが該当し，この部分パターンが期待するコンマ区切りの 1 個以上の式が含まれないためである．よって，再帰的なマクロの定義を可能にするためには，テンプレート内のマクロの入力を表すための新たな構文を導入しなければならない．

ここで，さらに 5 行目のパターンと 6 行目の下線部を見比べると，5 行目のパターンから *c1,* を除いた *c2, ...* 以降の部分パターンから想定される，テンプレート中のマクロの構文と下線部のマクロの入力が対応していることが分かる．このように，再帰的なマクロの定義に出現するマクロの入力はパターンの後半部分に対応するものが多い．そこで，本システムでは前述の試みをさらに一般化させた**接尾辞パターン**という考えを導入した．接尾辞パターンとは，あるパターンに対して，マクロ名以降の部分パターンのうち最も左にあるものを順番に 1 つずつ取り除いたものすべてのことをいう．そして，本システムでは，テンプレート内でのみ接尾辞パターンに対応したマクロの入力を許可した．例として，**Append** マクロの 2 つ目のパターンの接尾辞パターンを以下に示す．

```
Append , c2, ... to dom
Append c2, ... to dom
Append to dom
Append dom
```

図 7 の下線部は 2 つ目の接尾辞パターンに対応する入力と見なされ解析される．接尾辞パターンには，1 つ目の接尾辞パターンのように構文としてやや不自然なものも存在するが，これらはテンプレート内に出現するマクロの入力の構文解析にのみ利用するダミーのパターンである．実際のパターンにマッチしないものはマクロ展開時のパターンマッチで排除されるため問題はない．このような接尾辞パターンの導入により再帰的なマクロの定義も可能となった．

4. 非 Lisp 言語における実装の難しさ

Hygienic 構文マクロシステムは，安全かつ強力な記述力を持つマクロであるにもかかわらず，Scheme 以外の言語に実装された例はほとんどない．S 式のような一貫した構文構造を持たない非 Lisp 言語において Hygienic 構文マクロを実現するには，以下に述べる 2 つの課題を解決する必要がある．

4.1 マクロ構文の構文解析

1 つ目の課題はユーザが定義したマクロ構文の構文解析方法である．Scheme をはじめとする Lisp 言語のプログラムは，S 式という一貫した構文構造を持ち，それはユーザ定義のマクロ構文も同じである．マクロシステムが言語の構文構造を破壊することはなく，マクロ構文は従来の S 式パーザを用いて構文解析できる．一方，非 Lisp 言語の場合，マクロシステムはホスト言語にはない新たな構文をマクロとして追加する．そのため，従来のホスト言語のパーザではユーザ定義マクロを構文解析できない．

非 Lisp 言語における構文解析の複雑さの例として，図 7 のプログラムを構文解析することを考えよう．このプログラムには，通常の JavaScript 構文に加えて，マクロ定義とマクロ使用の構文が含まれる．これらの構文をそれぞれ，*JS*，*Definition*，*Macro* と呼ぶこととする．さらに，マクロ定義のテンプレートには，接尾辞パターンに対応したマクロの構文 *MacroSuf* と省略記号 ... の構文 *Dots* が含まれる．

これらの構文について，JS と Definition，Dots はあらかじめ形式が定まっており既知である．一方，Macro と MacroSuf はマクロ定義によって定められるため，一度マクロ定義を解析しなければそれらの構造を知る由はない．つまり，ユーザ定義マクロを含んだプログラムを構文解析

するためには，マクロ定義から動的に定められた Macro，MacroSuf を解析できるようパーザが拡張可能でなければならない．

次に各構文の出現について考える．JS と Macro はマクロ定義を除くプログラム全体に出現し，さらにマクロ定義のテンプレートにも出現する．また，テンプレートには MacroSuf，Dots も出現する．これらの構文の出現をまとめると，プログラムは 3 種類の文脈に分けることができる．マクロ定義以外のプログラム全体に相当する**プログラム文脈**，マクロ定義に相当する**マクロ定義文脈**，マクロ定義のテンプレートに相当する**テンプレート文脈**である．そして，これらの文脈に出現する構文はそれぞれ，(JS+Macro)，Definition，(JS+Macro+MacroSuf+Dots) である．

以上の考察から，マクロを含んだプログラムの構文解析には，パーザは拡張可能性に加えて，これらの 3 種類の文脈依存性への対応が求められる．

4.2　Hygienic 構文マクロ展開器の実装

2 つ目の課題は Hygienic 構文マクロ展開器の実装である．マクロ展開器は，パターンに合致する入力をテンプレートの形式に置き換える一種のプログラム変換器と見なせるが，そもそも Lisp は「プログラム = データ」な言語であるという特性がその実装を容易にしている．Lisp の場合，プログラムはすべて S 式という一貫した構文構造で表され，その S 式自体をデータとして扱うことができる．そのため，マクロ展開のようなプログラム変換も，S 式を直接操作することで容易に実現できる．一方で非 Lisp 言語は「プログラム $\neq$ データ」な言語である．プログラムはさまざまな構文要素からなる木を文字列で表したものであり，データとして扱うには構文木を操作しなければならない．また，構文木も一意ではないため，プログラム変換には不向きな言語である．よって，非 Lisp 言語では構文マクロの実現だけでも容易ではない．

さらに，Hygienic 性を保証したマクロ展開を実現しなければならない．マクロ展開の Hygienic 性は Kohlbecker ら [14] によって提唱され，その後 Clinger と Ree [5] によってマクロの参照透過性が追加された．あるマクロ展開が Hygienic であるためには，マクロ展開によって導入された変数束縛は同じマクロ展開によって導入された変数のみを捕捉し，また，マクロ展開によって導入された変数は同じマクロ展開によって導入された最も内側の変数束縛を参照し，そのような変数束縛がない場合には，マクロ定義内でその変数が出現した場所から最も内側にある変数束縛を参照するという条件を満たさなければならない．

この条件を満たすために，Hygienic マクロ展開器は束縛変数の名前を置換する α 変換を施すが，これは別のマクロから導入された同名の変数を区別するためのマーク付けと変数名を展開時の値に対応させるための置換によって行う [23]．Kohlbecker らが提案した Hygienic マクロ展開器の実装方式には，最悪時でプログラムの大きさについて 2 乗の計算量となる効率上の問題を抱えていた．この問題点を syntactic closure [4] にヒントを得て，Clinger らが提案したもの [5] が，今日の Scheme のマクロシステムの原型となった．仮に Scheme のマクロシステムを別のプログラミング言語に移植するために，その参照実装 [10] を利用するとしても，Hygienic マクロを用いてブートストラップ実装された 4,000 行のコードはマクロ展開により約 30,000 行のコードとなるため困難な道程だろう．Clinger らの提案を元としたコンパクトな実装は約 1,300 行の R4RS で記述されているが [11]，Scheme の文法の単純さと比較して，より複雑な構文構造を持つ一般的なプログラミング言語のための構文マクロシステムははるかに複雑で実装が困難なことが予想される．本稿は JavaScript 向けの Hygienic マクロシステムを提案しているものの，多様なプログラミング言語で Hygienic マクロシステムを利用するためには，その実装方式として可能な限り構文構造の複雑さに依存しない実装方式を考案することが重要である．

5.　提案技法

4 章で述べた課題に対し，Arai らはモジュール形式の動的拡張パーザと Scheme マクロ展開器を用いた実装技法を提案した [2]．ホスト言語で直接 Hygienic マクロ展開器を実装するのではなく，Scheme のマクロ展開器を用い，ホスト言語と S 式とを相互変換することによってマクロ展開を実現した．しかし，パーザに関しては，トップダウン構文解析法と演算子順位法を組み合わせた [16] 場当たり的なパーザを採用していたため，実装が複雑でマクロの記述力に乏しかった．

本研究では，解析表現文法（Parsing Expression Grammar, PEG）[9] を用いることによって，より表現力を高めた Hygienic マクロシステムを系統的に実装できることを示す．

5.1　解析表現文法

解析表現文法（PEG）は，解析的形式文法の一種で，非終端記号 A と解析表現 e からなる生成規則 $A \leftarrow e$ によって文法が定義される．その記法は文脈自由文法と正規表現を合わせた記法に似ており，構文規則と字句解析規則を合わせて記述できる．また，PEG によって表すことができる言語のクラスは和集合，共通部分，補集合について閉じているという特性を持つ．つまり，2 つの文法を合成して得られた文法も PEG であるといい換えることができる．

PEG の一番の特徴は，文脈自由文法での選択（|）にかわり，優先度付き選択（/）を用いる点である．`e1/e2` という解析表現は，`e1` の解析に成功すればそこで終了，失敗すれば `e2` を解析するという意味を持つ点が文脈自由文

1. マクロ定義の構文解析
2. マクロ構文の文法生成

（文法生成フェーズ）

3. 2 で生成した文法をホスト言語の文法に追加し，パーザ再構成
4. 3 で生成した拡張後パーザで構文解析
5. S 式へ変換
6. Scheme マクロ展開器で展開
7. ホスト言語へ変換

（マクロ展開フェーズ）

図 8　本システムによるマクロ展開の流れ

Fig. 8　The flow of a macro expansion by this system.

法との大きな相違点である．この優先度付き選択により，PEG には曖昧性がない．

また，PEG に強力な表現力を与える機能として，無制限の先読みとバックトラックがある．先読みには，`&` 述語と `!` 述語を用いる．`&e` は `e` の解析に成功すれば成功，失敗すれば失敗という意味で，`!e` はその逆であるが，これらはどちらも入力文字列を消費しない．

さらに，PEG は無制限の先読みが可能であるにもかかわらず，入力に対して線形時間で解析を行う packrat パーザに変換可能である [8]．

以上が PEG の特徴であるが，PEG が合成に関して閉じているという点が，拡張可能なパーザの実装において都合が良い．マクロ構文に対応した文法を後から追加することによって構文の拡張が表現でき，また，文法をモジュール化しておくことで，4.1 節で述べた 3 種類の文脈も表現しやすい．さらに，PEG において字句と構文の定式化が統合されているため，マクロ記述に現れる句読点やキーワードのような字句要素を構文記述に直接的に記述できる点はマクロシステムの実装にとって都合が良かった．これらの理由から，本研究では解析表現文法を採用した．

5.2　実装技法の概観

本実装方式では，文法記述にはすべて PEG を用いる．そして，ホスト言語の文法に動的に生成したマクロ構文の文法記述を追加し，それを基にパーザを再構成することで拡張可能なパーザを実現する．プログラム中のマクロ定義を解析した後，マクロ構文の文法を生成し，生成した文法をホスト言語の文法に追加して文法を拡張した後，パーザ生成系によってパーザを再構成する（**図 8**：1〜3）．

また，マクロ展開器については，Arai らの研究に習い，Scheme のマクロ展開器を利用する．マクロによって拡張されたホスト言語のプログラムをそれと等価な S 式へと変換し，Scheme マクロ展開器で展開を行った後，純粋なホスト言語のプログラムに逆変換する（図 8：4〜7）．

次節では，JavaScript 向け Hygienic 構文マクロシステム EX-JS の具体的な実装方法について述べる．

表 1　構文別に表した，事前に準備する文法とマクロ定義から動的に作成する文法

Table 1　Grammars which are prepared and ones which are dynamically generated from macro definitions, shown according to syntax.

文脈	事前に準備	マクロ定義から動的に作成
プログラム	JS	Macro
マクロ定義	Definition	–
テンプレート	(JS+Dots)	(Macro+Dots), (MacroSuf+Dots)

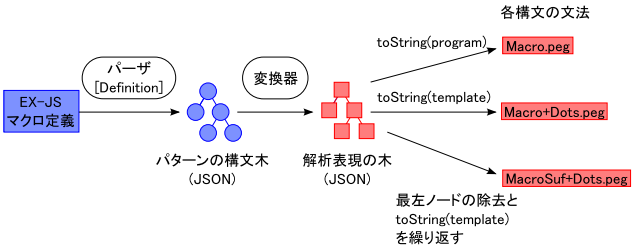

図 9　文法生成フェーズの処理の流れ

Fig. 9　The flow of processes at the grammar generation phase.

5.3　EX-JS の実装

本システムの実装において，パーザを構成するためのパーザ生成系には PEG.js[*1] を用いた．PEG.js は JavaScript 用のパーザジェネレータで，PEG に基づいた文法記述から対応する構文の packrat パーザを生成する．PEG.js では，PEG の解析表現として，意味述語の拡張が施されている．意味述語には `&{ }` と `!{ }` があり，どちらも `{ }` の中に JavaScript のコードを記述できる．そして，そのコードの返り値がそれぞれ true/false であれば成功という意味を持つ．この意味述語によって，本システムでは文脈に応じた処理の切り替えなどが簡潔に実現できている．さらに，解析表現に合致するコードが見つかったときに実行するアクションが記述できる．

PEG.js は JavaScript の仕様書 ECMA-262 [15] に従う文法記述を提供しているため，これを拡張して本システムの文法を作成した．ゆえに，本システムが扱う文法はすべて PEG.js 記法の文法であり，以下，特に言及しない限り，文法とは PEG.js 記法の文法記述のことである．

まず，EX-JS に用いる文法をそれが表す構文別に示すと**表 1** になる．この表からも分かるように，マクロ構文に関わらない既知の構文については事前に文法を用意しておき，マクロに関わる構文の文法だけを動的に生成する．本システムにおける文法生成フェーズを**図 9** に示す．Definition の文法から生成したパーザ（パーザ [Definition] と呼ぶ）を用いてマクロ定義を構文解析し，その結果を基に，Macro, (Macro+Dots), (MacroSuf+Dots) の文法を生成する．その詳細は 5.3.1 項で述べる．

次に，生成した 3 種類の文法と事前に準備した文法（JS,

[*1] http://pegjs.majda.cz/

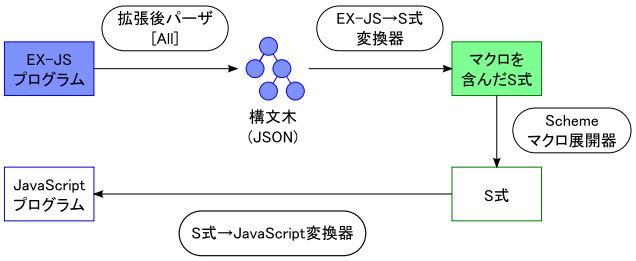

図 10　マクロ展開フェーズの処理の流れ

Fig. 10　The flow of processes at the macro expansion phase.

Definition，(JS+Dots) の文法）を合わせてパーザを再構成する．ここで再構成したパーザはすべての構文要素を解析可能であることから，拡張後パーザ [All] と呼ぶこととする．

そして，拡張後パーザ [All] を利用して**図 10** に示すマクロ展開フェーズに入る．ここでは，EX-JS プログラムを拡張後パーザ [All] で構文解析し，その解析結果をマクロを含んだ S 式へと変換，そして Scheme マクロ展開器を用いて展開後，S 式から JavaScript へのコード変換を行う．S 式への変換部は 5.3.2 項で，JavaScript への変換部は 5.3.3 項で詳しく述べる．

5.3.1　マクロ構文の文法生成

文法生成フェーズでは，まず初めにマクロ定義の構文解析を行うが，マクロ定義のうち，マクロの構文形式を表すものはパターンである．よって，本システムはマクロ定義を構文解析して得られた構文木のうち，パターンに相当する部分木からマクロ構文の文法を生成する．

パターンの構文木は**図 11** に示す抽象構文で表される．IdentifierVar，ExpressionVar，StatementVar は各マクロ変数を表すノードである．また，Paren，Brace，Bracket，RepBlock はそれぞれ `( )`, `{ }`, `[ ]`, `[# #]` に，Repetition は繰返しを表す $\langle p \rangle \ldots$ に対応するノードである．これらのノードから構成される構文木はごく自然に文法の断片を表す解析表現の木に変換できる（**図 12**）．

このようにして得られた解析表現の木に対して，各ノードを実際の解析表現を表す文字列へ変換したものが目的の文法である．このとき，文字列への変換において，変換関数の引数に目的の文脈を表す program か template を与えると，それに応じてプログラム文脈用，テンプレート文脈用の解析表現を生成する．**表 2** に各ノードと生成される解析表現との対応を示す．この中で `MacroIdentifier` や `AssignmentExpression` など，アルファベットの大文字で始まるものは各構文要素を表す非終端記号である．また，`__` は 0 個以上の空白を表す非終端記号である．なお，実際に生成される解析表現には解析に成功したときに実行するアクションの記述も含まれるが，ここでは簡単のため省略している．

PEGLiteral と PEGKeyword から生成される解析表現に

$$
\begin{aligned}
\mathit{pattern} &::= \text{Pattern}(\mathit{name}, \mathit{ps}) \\
\mathit{ps} &::= \text{Nil} \mid \text{Cons}(\mathit{p}, \mathit{ps}) \\
\mathit{p} &::= \text{Literal}(\mathit{ltype}, \mathit{value}) \\
&\mid \text{IdentifierVar}(\mathit{name}) \\
&\mid \text{ExpressionVar}(\mathit{name}) \\
&\mid \text{StatementVar}(\mathit{name}) \\
&\mid \text{Keyword}(\mathit{name}) \\
&\mid \text{Paren}(\mathit{ps}) \mid \text{Brace}(\mathit{ps}) \\
&\mid \text{Bracket}(\mathit{ps}) \mid \text{RepBlock}(\mathit{ps}) \\
&\mid \text{Repetition}(\mathit{p}) \\
\mathit{ltype} &::= \text{NullLiteral} \mid \text{BooleanLiteral} \mid \text{NumericLiteral} \\
&\mid \text{StringLiteral} \mid \text{RegularExpressionLiteral}
\end{aligned}
$$

図 11　パターンを表す構文木の抽象構文

Fig. 11　The abstract syntax of a syntax tree which expresses a pattern.

$$
\begin{aligned}
T_{\mathrm{P}}[\text{Pattern}(\mathit{name}, \mathit{ps})] &= \text{Cons}(\text{PEGMacroName}(\mathit{name}), T_{\mathrm{P}}[\mathit{ps}]) \\
T_{\mathrm{P}}[\text{Nil}] &= \text{Nil} \\
T_{\mathrm{P}}[\text{Cons}(\mathit{p}, \mathit{ps})] &= \text{Cons}(T_{\mathrm{P}}[\mathit{p}], T_{\mathrm{P}}[\mathit{ps}]) \\
T_{\mathrm{P}}[\text{Literal}(\mathit{ltype}, \mathit{value})] &= \text{PEGLiteral}(\mathit{ltype}, \mathit{value}) \\
T_{\mathrm{P}}[\text{IdentifierVar}(\mathit{name})] &= \text{PEGIdentifier} \\
T_{\mathrm{P}}[\text{ExpressionVar}(\mathit{name})] &= \text{PEGExpression} \\
T_{\mathrm{P}}[\text{StatementVar}(\mathit{name})] &= \text{PEGStatement} \\
T_{\mathrm{P}}[\text{Keyword}(\mathit{name})] &= \text{PEGKeyword}(\mathit{name}) \\
T_{\mathrm{P}}[\text{Paren}(\mathit{ps})] &= \text{PEGParen}(T_{\mathrm{P}}[\mathit{ps}]) \\
T_{\mathrm{P}}[\text{Brace}(\mathit{ps})] &= \text{PEGBrace}(T_{\mathrm{P}}[\mathit{ps}]) \\
T_{\mathrm{P}}[\text{Bracket}(\mathit{ps})] &= \text{PEGBracket}(T_{\mathrm{P}}[\mathit{ps}]) \\
T_{\mathrm{P}}[\text{RepBlock}(\mathit{ps})] &= \text{PEGRepBlock}(T_{\mathrm{P}}[\mathit{ps}]) \\
T_{\mathrm{P}}[\text{Repetition}(\mathit{p})] &= \text{PEGRepetition}(T_{\mathrm{P}}[\mathit{p}])
\end{aligned}
$$

図 12　パターンを表す構文木のノードから解析表現の木のノードへの変換規則

Fig. 12　Conversion rules from a node of the syntax tree which expresses a pattern to a node of parsing expression tree.

は意味述語が含まれている．これは，パターン中のリテラルやキーワードに対応するマクロ使用の入力には，それと等価な値（キーワードの場合は識別子名）のリテラル，キーワードが期待されることから，その等価性の判定のため，ここで意味述語を用いている．PEGLiteral の場合，*ltype* には図 11 にあげた各リテラルの型を表す非終端記号名が入る．よって，この解析表現は型が同じでかつ値が等しいリテラルの入力を受理する．なお，この解析表現の中で用いている `v:` は PEG.js 記法に含まれるラベルという解析表現で，これを付加することにより意味述語の中で `v` を使って解析結果にアクセスすることができる．同様に，PEGKeyword から生成される解析表現はキーワードに該当し，かつその名前が等しい入力を受理する．

この表が示すように，プログラム文脈とテンプレート文

表 2　解析表現を表すノードと各ノードから生成される解析表現
Table 2　Nodes which express parsing expressions and parsing expressions which are generated from each node.

解析表現を表すノード	生成される解析表現
Nil	
Cons(*e*, *es*)	*e* `__` *es*
PEGLiteral(*ltype*, *value*)	`v:`*ltype* `&{ return` `  eval(v.value) ===` *value*`; }`
PEGIdentifier	`MacroIdentifier`
PEGExpression	`AssignmentExpression`
PEGStatement	`Statement`
PEGKeyword(*name*)	`v:MacroKeyword &{ return` `  v.name === "`*name*`"; }`
PEGParen(*es*)	`"(" __` *es* `__ ")"`
PEGBrace(*es*)	`"{" __` *es* `__ "}"`
PEGBracket(*es*)	`"[" __` *es* `__ "]"`
PEGRepBlock(*es*)	*es*

PEGRepetition(*e*)	`//` プログラム文脈用
	`(`*e* `(__` *e*`)*)?`
	`//` テンプレート文脈用
	`(`*e* `(__` *e*`)* (__ "..")?)?`

PEGMacroName(*name*)	`//` プログラム文脈用 "*name*" `!IdentifierPart` `//` テンプレート文脈用 `&{ return templateContext; }` "*name*" `!IdentifierPart`

脈で生成される解析表現が異なるノードは PEGRepetition と PEGMacroName である．PEGRepetition は，パターン中で ... があった位置に，実際の入力として ... を許すかどうかが文脈の違いである．

一方，PEGMacroName から生成される解析表現を見ると，テンプレート文脈用の解析表現には意味述語が用いられているが，これがこの解析表現がテンプレート文脈のみでしか解析の対象とならないようにするための条件節となる．マクロ構文は必ずマクロ名から始まるため，パーザがテンプレート文脈用のマクロ構文の解析表現を解析しようとすると必ずこの意味述語に到達する．そして，この意味述語の中の `templateContext` という変数は，入力文字列におけるパーザ解析地点がテンプレート文脈かどうかを表しているのだが，その値が true であれば解析に成功し，次の "*name*" 以降の解析表現について解析を進めることができるという仕組みになっている．このように，意味述語を用いることで，文脈に応じて解析するか否かの簡潔な切り替えが可能となった．なお，ここで "*name*" の後に，`!` 述語を用いて識別子を構成する文字があってはならないことを意味した `!IdentifierPart` という解析表現があるが，これはマクロ名ではない識別子を誤って解析しないためのものである．もしこの解析表現がないと，たとえば M というマクロが定義されている場合，M1 という識別子はマクロ名 M と式 1 として構文解析されてしまう．よって，このような構文解析の間違いを避けるために，`!` 述語を用いている．

$T_{\text{EX-JS}}$[**expression** *macro* {
　　keyword: *key*, ...;
　　{ *pattern* => *template* } ... }]
= (**define-syntax** *macro-Macro*
　(**syntax-rules** (*V-key* ...)
　　($T_{\text{EX-JS}}$[*pattern*] $T_{\text{EX-JS}}$[*template*]) ...))
$T_{\text{EX-JS}}$[*macro p* ...] = (*macro-Macro* $T_{\text{EX-JS}}$[*p*] ...)
$T_{\text{EX-JS}}$[[# *p* ... #]] = ($T_{\text{EX-JS}}$[*p*] ...)
$T_{\text{EX-JS}}$[**function** *f* (*v*, ...) { *s* ... }]
= (**define** (*V-f V-v* ...) $T_{\text{EX-JS}}$[*s*] ... #\@)
$T_{\text{EX-JS}}$[**function** (*v*, ...) { *s* ... }]
= (`"JS" "function"` #*nul*
　(**lambda** (*V-v* ...) $T_{\text{EX-JS}}$[*s*] ... #\@))
$T_{\text{EX-JS}}$[*v*] = *V-v*
$T_{\text{EX-JS}}$[**var** *v* = *e*] = (**define** *V-v* $T_{\text{EX-JS}}$[*e*])
$T_{\text{EX-JS}}[c] = \begin{cases} (\texttt{"JS" "string" "}c\texttt{"}) & (c \text{ が StringLiteral のとき}) \\ (\texttt{"JS" "const" "}c\texttt{"}) & (\text{上記以外}) \end{cases}$
$T_{\text{EX-JS}}$[[*e*, ...]] = (`"JS" "array"` $T_{\text{EX-JS}}$[*e*] ...)
$T_{\text{EX-JS}}[e_1 \oplus e_2]$ = (`"JS" "binary" "`$\oplus$`"` $T_{\text{EX-JS}}[e_1]$ $T_{\text{EX-JS}}[e_2]$)
$T_{\text{EX-JS}}$[**if** (*e*) s_1 **else** s_2]
= (`"JS" "if"` $T_{\text{EX-JS}}[e]$ $T_{\text{EX-JS}}[s_1]$ $T_{\text{EX-JS}}[s_2]$)
$T_{\text{EX-JS}}[f(e, ...)]$ = (`"JS" "funcCall"` *V-f* $T_{\text{EX-JS}}$[*e*] ...)

図 13　EX-JS から S 式への変換規則の例
Fig. 13　Examples of conversion rules from EX-JS to S-expression.

さらに，解析表現の木に対して最左ノードの除去とテンプレート文脈用の解析表現の生成を繰り返し，接尾辞パターンに対応した文法を得る．

こうしてパターンごとに生成した各マクロ構文の文法は，パターンが定義された順に優先度付き選択（`/`）を用いて連結される．そのため，本システムではパターンを定義する順序がマクロの構文解析に影響を与える．この点については 8.2 節で述べる．以上が文法生成の流れである．

5.3.2　S 式への変換

拡張後パーザ [All] による構文解析で得られた構文木を，ここでは S 式へと変換する．ここで注意すべきことは，EX-JS 側でのプログラムの意味を S 式側でも正しく保持しておくことである．また，Hygienic マクロ展開を実現するために，EX-JS 側のマクロ定義，マクロ使用，変数束縛，スコープを S 式側でも再現しなければならない．そこで，これらに関するコードは Scheme の変数の扱いに関する特殊形式に変換する．そして，それ以外の構文は，EX-JS と S 式との間で意味の違いが生じることを避けるため，タグ付けを用いて変換する．代表的な変換例を**図 13** に示す．この図においては，式マクロの例のみあげているが，文マクロも定義・使用ともに式マクロと同じ変換規則に従う．また，マクロ使用のマクロ名以降の部分や `[# #]` の中身はその入力形式が不明なため *p* としている．さらに，関数の変換規則に出現する #\@は，関数の本体が空にならないために使用している記号で，特に意味を持つものではない．

```
(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))
```

図 14　図 6 をマクロを含んだ S 式に変換した結果
Fig. 14　A result of converting Fig. 6 into S-expression which contains macros.

この図が示すように，タグ付けは文字列によって行う．これは，Scheme ではマクロ定義のパターン中で同じ変数を使うことが禁止されており，もしタグに識別子を用いると，マクロ定義中に出現したタグが変数と見なされ，変数の重複が起こってしまう可能性があるためである．タグは文字列ではなく数値でもかまわないが，可読性から文字列を用いた．

変換の例として，**図 14** に図 6 の **let** マクロの定義と使用を変換した S 式を示す．この S 式は，タグが文字列であるために，Scheme のプログラムとしては構文的に間違ったものであり評価できない．しかし，マクロ展開において必要なものは，プログラムの意味が保持されていることであり，式の評価は不要である．加えて，Scheme マクロ展開器は式の評価を行わないため，文字列タグによる変換は適切な手法であるといえる．

5.3.3　JavaScript への変換

マクロ展開後の S 式から JavaScript への変換は，主に S 式中のタグや特殊形式に基づいて逆変換を行うだけである．マクロ展開後の S 式は EX-JS に特有のマクロに関わるタグは含まないため，逆変換を行うと純粋な JavaScript のコードが得られる．

例として，まず図 14 をマクロ展開した結果を**図 15** に示す．本システムは Scheme の処理系 Ypsilon[*2] のマクロ展開器を用いているが，この図が示すように変数衝突の原因となりそうなローカル変数の名前が変更され，適切に Hygienic マクロ展開が施されていることが分かる．

[*2] http://code.google.com/p/ypsilon/

```
(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\x60;200* V-a\x60;200*)
      ("JS" "return"
       ("JS" "funcCall"
        ("JS" "propAccess" V-console "log")
        V-where\x60;200* V-a\x60;200* V-b))
      #\@))
   ("JS" "string" "local")
   ("JS" "const" "100"))
  ("JS" "funcCall"
   ("JS" "propAccess" V-console "log")
   V-where V-a V-b))
```

図 15　図 14 をマクロ展開した結果
Fig. 15　A result of a macro expansion of Fig. 14.

```
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));
```

図 16　図 6 をマクロ展開した結果最終的に得られるコード
Fig. 16　The code finally obtained as a result of a macro expansion of Fig. 6.

この S 式を JavaScript へ変換した結果が**図 16** である．Ypsilon のマクロ展開器は α 変換時に，JavaScript では扱えないような名前を生成するため，それを JavaScript が扱えるものに変更する必要がある．そのため，S 式中で *V-where\x60;200** であった変数が *where_200_* と変更されている．以上のようにして本システムは Hygienic マクロ展開を行う．

6.　EX-JS の記述力を支えるその他の機能

3 章では，EX-JS のマクロシステムとしての中核の機能を紹介した．ここでは，EX-JS がより記述力の高いマクロを実現するために提供している機能を紹介する．

6.1　シンボル

以下は，出発地から目的地までの移動経路を示す記述であるが，この日本語のような記述も，EX-JS で定義したマクロを用いて記述できる．

```
乗り換え経路 大岡山 から 羽田 まで 電車 で
             羽田 から 奄美大島 まで 飛行機 で 行きます
```

これを展開すると，以下の経路情報を示した配列を表すコードが得られる．

```
1 expression 乗り換え経路 {
2   symbol: A, B, T;
3   keyword: から, まで, で, 行きます;
4   { 乗り換え経路
5     [# A から B まで T で #] ... 行きます
6     => [{ from: A, to: B, by: T }, ...] }
7 }
```

図 17　乗り換え経路マクロの定義
Fig. 17　A definition of 乗り換え経路 macro.

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

図 18　区切り記号に and を用いた **Append** マクロの定義と使用
Fig. 18　A definition of **Append** macro using “and” as punctuation marks and its usage.

```
[ { from: "大岡山", to: "羽田", by: "電車" },
  { from: "羽田", to: "奄美大島", by: "飛行機" } ];
```

このコードと展開前のコードを見比べると，展開前のコードでは文字列データとして扱われる大岡山や羽田に引用符を用いていないことが分かる．このような記述を可能にするのがシンボル機能である．

乗り換え経路マクロの定義を**図 17** に示す．ここで，**symbol** 宣言子によってマクロ変数 *A*，*B*，*T* が宣言されていることが分かる．これら **symbol** 宣言されたマクロ変数に対応する入力は，識別子として構文解析された後，内部的に文字列として変換され処理される．こうして，引用符を用いない簡潔な文字列の記述が可能となった．

6.2　区切り記号

次に，88 ページの図 6 や 89 ページの図 7 の例では，マクロ使用の中で *Title*, *Menu*, *Content* のように繰返し要素を区切るための記号としてコンマを用いた．これは，マクロ定義のパターンにおいて，省略記号 ... の前にコンマを記述し，繰返しの区切り記号がコンマであることを明記したためである．通常，JavaScript では式の区切りにコンマを用いるため，それをこの例にも採用した．

しかし，実際にマクロを記述するときにコンマ以外の区切り記号を用いたいこともある．そこで本システムでは，マクロ変数やキーワードでない識別子・記号を区切り記号として使用することを許可した．例として，図 7 の **Append** マクロの区切り記号に and を用いると，**図 18** のようになる．なお，この例では省略記号の前の区切り記号は 1 つであるが，複数の区切り記号を記述することも可能である．

表 3　区切り記号に関連するノードから生成される解析表現
Table 3　Parsing expressions which are generated from nodes related to punctuation marks.

解析表現を表すノード	生成される解析表現
PEGPunct(*punct*)	`"`*punct*`"`

PEGRepetition(*e*, *es*)	// プログラム文脈用
	(*e* (__ *es* __ *e*)*)?
	// テンプレート文脈用
	(*e* (__ *es* __ *e*)* (__ *es* __ "...")?)?

```
1 (begin
2   (define-syntax Append-Macro
3     (syntax-rules (V-to)
4       ((_ V-c1 V-to V-dom) **省略**)
5       ((_ V-c1 V-c2 ... V-to V-dom)
6        (Append-Macro V-c2 ... V-to **省略**))))
7   (begin
8     (define V-pageBody
9       (Append-Macro V-Title V-Menu V-Content
10       V-to
11       ("JS" "funcCall" V-jQuery **省略**)))))
```

図 19　図 18 を S 式へ変換した結果
Fig. 19　A result of converting Fig. 18 into S-expression.

本システムにおける区切り記号は，マクロ使用の見栄えを良くし，記述力を高めるためのものであり，キーワードのようにマクロ展開の中で意味を持つものではない．よって，構文解析時にはその入力が正しいか検査されるが，パターンマッチ時には無視される仕組みとなっている．

区切り記号はパターンを表す抽象構文木においてノード PunctuationMark(*punct*) で表される．また，92 ページの図 11 では，繰返しを表す部分パターンに相当するノードを Repetition(*p*) としていたが．実際には繰返し要素 *p* と区切り記号の列 *ps* で構成される Repetition(*p*, *ps*) となる．そして，これらのノードを解析表現のノードへ変換すると PEGPunct(*punct*)，PEGRepetition(*e*, *es*) となり，これらから生成される解析表現は**表 3** のようになる．このようにパターンから生成される文法の中には，区切り記号を解析するための解析表現が含まれるため，構文解析時には区切り記号に対応する入力が検査される．

一方，前述のように S 式への変換時にはこれらの区切り記号は除去される．たとえば，図 18 の **Append** マクロの定義と使用を S 式へ変換すると**図 19** のようになる．これにより，区切り記号はパターンマッチの対象とはならない．

以上のシンボルや任意の区切り記号を用いることで，本システムでは一見プログラミング言語ではないようなマクロの記述も可能となっている．

7.　評価

5 章では，本研究が提案する Hygienic 構文マクロシステ

表 4　EX-JS の実装に要したコード行数
Table 4　Lines of code required for an implementation of EX-JS.

プログラムが行う処理	行数
EX-JS の文法（PEG.js）	595
図 8：1–4 の構文解析，文法生成，パーザ拡張（JavaScript）	689
図 8：5 の EX-JS → S 式変換（Racket）	208
図 8：6–7 のマクロ展開，S 式→ JavaScript 変換（Scheme）	439
その他（ユーティリティ関数など）	43
合計	1,974

ムの実装技法について述べたが，ここでは，その評価として，EX-JS の実装にかかったコード量とマクロ展開にかかる処理時間について述べる．

7.1　コード量

本システムは，PEG.js 記法の文法と JavaScript，Racket [*3]，Scheme，そして，すべてのプログラムをまとめて実行するためのシェルスクリプトのコードからなる．これらのコード行数を**表 4** に示す．ここで，EX-JS の文法とは PEG.js が提供する ECMA-262 の文法（1,532 行）を拡張するために，差分的に記述した文法のことである．ただし，本システムにおいて ECMA-262 の文法と EX-JS の文法を用いると，文法の合成によって同じ非終端記号に対する生成規則を上書きすることになり，パーザ再構成において非効率となる．そのため，実際には ECMA-262 を上書きした 1824 行の文法を利用している．

また，本システムは EX-JS → S 式変換とマクロ展開，S 式→ JavaScript 変換において，それぞれ Racket と Scheme を用いている．Racket は Scheme から派生した言語であるが，これらの 2 種類の言語を用いたのは，Racket には JSON を扱うライブラリがあり，EX-JS → S 式変換に適していたこと，そして，Racket のマクロ展開では展開後の結果が扱いにくく，Scheme の処理系 Ypsilon のマクロ展開が最も扱いやすい結果を出し，その後の S 式→ JavaScript 変換に適していたことが理由である．

以上の実装は表 4 が示す 2,000 行に満たないコードから構成されている．すでに述べたように，提案した実装の中心的な役割は，PEG に基づいた構文解析器の自動生成と，JavaScript と S 式の間の相互翻訳である．このうち Hygienic マクロ展開に関わるのは，相互変換において変数の束縛と使用に関わる情報を正しく保持することであり，変数に関わらない言語機構はその構造を S 式のデータ構造として受け渡しているのみである．本提案の後半は Arai らの研究 [2] を踏襲しているが，前半の構文の扱いを PEG を用いた手法に置き換えることで，より自由なマクロの記述を可能とする強力なマクロ機能を提供するとともに，実装はコンパクトで系統的な手法となった．

[*3] http://racket-lang.org/

7.2　マクロ展開にかかるコスト

本システムは，JavaScript の処理系に Node.js [*4]，パーザ生成系に PEG.js，Racket の処理系に Racket，Scheme の処理系に Ypsilon を用いている．Scheme の処理系にはいくつかあるが，Ypsilon が R6RS をサポートしており，かつ，先にも述べたようにマクロ展開の結果が見やすく，その後の変換において扱いやすい形であるために，これを採用した．

本システムにおけるマクロ展開のコストとして，これまでに例として出てきた図 3（unless.js），図 6（let.js），図 7（append.js），図 17 に本文中のマクロ使用のコードを加えたもの（path.js），図 18（append-and.js）をマクロ展開したときの処理時間を**表 5** に示す．比較対象として空ファイル（empty.js）のマクロ展開も行った．

実行環境は以下のとおりである．

- CPU: 2.53 GHz Intel Core 2 Duo
- Memory: 4 GB 1067 MHz DDR3
- OS: Mac OS X 10.6.8
- Node.js v0.6.15
- PEG.js v0.7.0
- Racket v5.2
- Ypsilon 0.9.6-update3

いずれのファイルも 7 秒以上要しており，その大部分を占めているのはパーザ再構成である．empty.js の場合，マクロ構文にかかわる文法は生成しないため，事前に準備した JS，Definition，(JS+Dots) の文法からパーザを生成するが，それだけでも約 6.1 秒かかることが分かる．ちなみに，PEG.js が提供する ECMA-262 の文法からパーザを生成するのにかかる時間は約 2.1 秒である．前者の文法の行数は後者の 1.2 倍ほどであるにもかかわらず，そのパーザ生成には 3 倍もの時間を要する．本システムにおいて，パーザ生成にこれほど時間がかかってしまう原因は文法の記述方法にあると考えられるが，この点については目下調査中である．

また，let.js と path.js が他に比べて大きなパーザ生成時間を要しているが，これらのマクロが他と異なる点は `[# #]` によって繰返し範囲を指定し，かつ，その繰返し要素の中でキーワードを用いていることである．**図 20** が示すように，`[# #]` 中でキーワードを使用するとその個数の 2 乗に比例してパーザ生成時間が増加する．よって，このことが他のマクロよりもパーザ生成に時間がかかってしまった原因の 1 つであると考えられる．

今回の実装ではパーザ生成系に PEG.js を利用したが，先に示したように PEG.js には `[# #]` 中のキーワード等パーザ生成においてボトルネックとなる文法記述がいくつかある．その究明には至っていないが，これは記述方法を

[*4] http://nodejs.org/

表 5　マクロ展開にかかる処理時間
Table 5　Processing time for a macro expansion.

処理系	処理	時間（秒）					
		empty.js	unless.js（図 3）	let.js（図 6）	append.js（図 7）	path.js（図 17）	append-and.js（図 18）
Node.js	マクロ定義の構文解析	0.009	0.034	0.044	0.046	0.054	0.059
	マクロ構文のための文法生成	0.000	0.001	0.001	0.001	0.001	0.001
	パーザ再構成	6.140	6.276	8.364	6.627	15.357	6.671
	拡張パーザでの構文解析	0.008	0.020	0.023	0.020	0.025	0.019
	起動時間	0.049	0.049	0.049	0.049	0.048	0.048
	ファイル入出力など	0.266	0.236	0.245	0.250	0.278	0.237
Racket	EX-JS → S 式変換	0.009	0.020	0.019	0.017	0.019	0.017
	起動時間	0.072	0.072	0.072	0.072	0.072	0.072
	ファイル入出力など	0.451	0.449	0.448	0.448	0.449	0.448
Ypsilon	マクロ展開と S 式→ JavaScript 変換	0.000	0.003	0.002	0.001	0.002	0.001
	起動時間	0.030	0.030	0.030	0.030	0.030	0.030
	ファイル入出力など	0.038	0.037	0.038	0.038	0.038	0.038
	合計	7.072	7.227	9.335	7.599	16.373	7.641

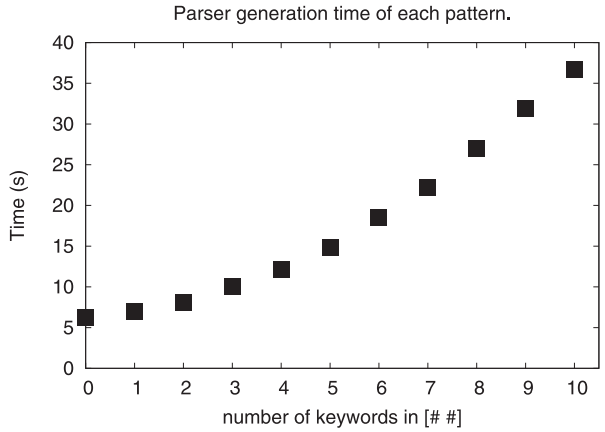

図 20　パターン **macro** [# #] ... の [# #] 中のキーワードの数とパーザ生成時間の関係
Fig. 20　The relation between the number of keywords in [# #] of the pattern **macro** [# #] ... and parser generation time.

改善することに加え，パーザ生成系を別のものに変更することでも解決可能であると考えている．また，本システムでは文法は差分的に生成しているものの，パーザの生成は差分的に行えていない．差分的に定義した文法のみからのパーザの拡張が可能になれば，事前に準備した文法との合成は不要になり，各パーザ再構成に要する時間は empty.js に要した約 6.1 秒を引いた値となる．これが実現できれば，マクロ展開のコストは大幅に軽減される．

現実装における展開コストは，頻繁に編集と実行を繰り返すような開発スタイルがとられる JavaScript において非常に大きなものである．しかし，本システムにはパーザキャッシュ機能があり，一度生成したパーザはすべて保存されるため，マクロの定義が変わらない限り，2 回目以降はパーザの再構成は行われない．実際，1 回目の展開で最も時間のかかった path.js も 2 回目は 1.1 秒程で完了した．よって，本システムにおいては，マクロ定義を終えた後，その他の部分について編集と実行を繰り返すという開発スタイルが望ましいと考えられる．

8. 議論

ここまでで，本システムの実装方法や仕様について見てきた．ここでは，本システムが与える制約やその記述力について述べる．

8.1 S 式への変換による制約

本システムは，マクロ展開の過程でプログラムを S 式に変換するという都合上，EX-JS で記述できる JavaScript の構文に以下の制約を与える．

(1) 変数宣言・関数定義がローカルスコープの先頭またはグローバルスコープの中にあること
(2) **with** 文が使われていないこと

本来，JavaScript では変数宣言・関数定義の位置に制約はない．しかし，本システムはそれらを Scheme の **define** 式に変換しており，Scheme では **define** 式はローカルスコープの先頭かグローバルスコープの中になければならないという制約がある．よって，この Scheme 側での構文規則を守るため，(1) の制約を設けた．

また，JavaScript の **with** 文は特殊なスコープ規則を持っており，変数参照には実行時に動的に生成したスコープを，変数束縛にはその外側のスコープを用いる．これを静的な解析のみで Scheme 側で再現することは難しい．そのため，制約 (2) を設けた．

ただし，これらの制約は，実装方式の改良によって不要となる可能性もある．S 式変換前に構文解析だけでなく意味解析も行い，適切なプログラム変換を施せば，制約のない JavaScript を対象とすることもできるだろう．しかし，

```
1 statement unless {
2   expression: C;
3   statement: S1, S2;
4   keyword: else;
5   { unless (C) S1 else S2
6     => if (!C) S1 else S2 }
7   { unless (C) S1 => if (!C) S1 }
8 }
```

図 21　**else** 節を持つ **unless** マクロの定義
Fig. 21　A definition of **unless** macro with **else** clause.

実際のところ，マクロ展開前に式の順番を変えるなどのプログラム変換を施すことは，マクロ展開後の結果に影響を及ぼす恐れがあり，その実装は難しい．

そのため，本システムはこれらの対応を見送った．また，JavaScript の良いパーツ，悪いパーツについて述べた [6] の中で，**with** 文は悪いパーツに分類され，さらに変数宣言・関数定義もスコープの先頭で定義されるべきだという記述もあるため，これらの制約を設けても JavaScript プログラムの表現力には問題ないと判断した．

8.2　マクロの構文解析

本システムのような非 Lisp 言語におけるマクロシステムでは，マクロの構文解析が曖昧になる場合がある．たとえば，87 ページの図 3 であげた **unless** マクロを **if** 文と同じ文法を持つように定義した場合，**else** 節の結合性に関わる問題が生じる．

これに対し本システムでは，5.3.1 項でも触れたように，パターンから生成された文法は定義された順に優先される選択肢となるため，図 **21** のように **else** 節を持つパターンを先に書けば **else** 節は最も近い **unless** と結合するよう構文解析される．一方で，2 つのパターンの定義の順序を入れ替えると **else** 節を持つ **unless** マクロの使用は構文解析に失敗する．なぜなら，優先度の高い **else** 節を持たないパターンによって，マクロ使用の **else** 節の前までの部分が受理されてしまい，後続の **else** から開始する文というものはマクロ構文にも JavaScript の構文にも含まれないからだ．このように，本システムでは先に定義されたパターンから構文解析を行うことから，複数のパターンを持つマクロの場合，優先度の高いパターンから順に定義しなくてはならない．

なお，この例からも分かるように，本システムでは多くの場合において，マクロ使用に該当するパターンは構文解析時に決定付けられる．すなわち，構文解析がパターンマッチと同等の役割を担う．そのため，パターンマッチに失敗するマクロ使用は構文解析時にエラー検出されることとなる．

8.3　記述力の制限

これまでの例で見てきたように，本システムのマクロは比較的自由度の高い記述が可能である．括弧記号の対応など最低限の決まりはあるが，従来の JavaScript 構文にとらわれない自由なマクロの定義ができる．しかし，その自由さゆえに，マクロ使用の終端が分かりにくいという欠点があり，マクロ使用を明示的に括弧で囲わなければならない場合がある．

たとえば，88 ページの図 6 で定義した **let** マクロの使用を $*$ 演算子の被演算子とすることを考えよう．**let** マクロを用いて変数 x，y を 1，2 に束縛し，$x+y$ の値を計算した後，その値に 3 をかける．しかし，以下のように記述すると，**In** 節の式が $x+y*3$ と構文解析されてしまい，展開後のコードを実行すると，本来期待した 9 ではなく 7 という結果が得られる．

```
let x = 1, y = 2 In x + y * 3;
```

これが正しく解析されるためには，以下のようにマクロ使用を括弧で囲わなければならない．

```
(let x = 1, y = 2 In x + y) * 3;
```

この例のように，マクロ使用の範囲が曖昧な入力が与えられると，パーザは貪欲に解析を続けるため，本来意図したものと異なる結果をもたらす場合がある．したがって，終端が自明でないときにはマクロ使用の範囲を明記しなければならない．一方で，95 ページの図 17 の乗り換え経路マクロのように終端が「**行きます**」と自明で曖昧性をもたらす危険がない場合には括弧で囲う必要はない．

この問題は，マクロ構文の形式が関数適用のような形に限られたマクロシステムであれば生じなかったものである．しかし，本システムはより柔軟なマクロの記述を優先した．とはいえ，すべての場合ではないが，マクロ使用を括弧で囲わなければならないとなると，関数適用風のマクロを記述しているのと同じである．よって，本システムは記述力の高いマクロではあるが，あくまでそれはマクロを使う際に曖昧性がない場合についてであり，それ以外は括弧記号の入力により記述力は制限される．この問題への対処に向けて，本システムにおけるマクロがどのような場合に曖昧性を生むかを調査することは，今後の課題の 1 つである．

8.4　ローカルマクロの実現可能性

Scheme では **define-syntax** のかわりに **let-syntax** を用いることで，**let-syntax** に与えられた式の中でだけ有効なローカルマクロを定義することができる．一方，現実装における本システムでは，このようなローカルマクロを定義するための仕組みは提供していない．

ローカルマクロの定義を可能にするためには，マクロが有効となる範囲でだけ構文解析を行うようなローカルなパーザが必要になる．これを本システムで実現するには，意味述語を用いてマクロの有効範囲を示したマクロ構文の

文法を生成し，それを基にパーザを再構成すればよい．これは，意味述語を用いて文脈による処理の切り替えを行ったのと同様のことである．よって，マクロの有効範囲を明示的に示す **let-syntax** に相当する新たなマクロ宣言子を導入すれば，ローカルマクロの実現は可能であろう．

8.5　マクロ定義の位置

本システムでは，マクロ定義はグローバルスコープの中に記述することとしている．このため，マクロ定義の順序に制約はなく，R6RS のマクロの仕様と同様にプログラムのトップレベルでは定義と式・文が交互に出現可能である．ただし，定義したマクロの有効範囲は，R6RS ではマクロ定義より後ろのプログラムのみであるのに対して，本システムではファイル全体である．この違いはパーザに起因するものであるが，Scheme のマクロ展開器を用いてマクロ展開している本システムにおいて，これはバグを引き起こす要因になりかねない．そのため，本システムでは暗黙的にプログラムの先頭でマクロを定義することが望まれる．ただし，前節で述べたローカルなパーザを実装すれば，この問題の修正は容易である．

9.　関連研究

非 Lisp 言語に Hygienic 構文マクロシステムを導入した最初の例はオブジェクト指向言語の一種 Dylan [18] である．Dylan は，**定義マクロ**，**文マクロ**，そして**関数マクロ**を通してそれぞれ新たな定義，文，そして式の定義を許している．この点で，本研究における文マクロと式マクロに類似している．Dylan は内部では Lisp の S 式を若干拡張したスケルトン形式と呼ばれるある種のパターン言語によってプログラムを表現しており，構文解析器とマクロ展開器はスケルトンの構造に沿って実装されている．このため Dylan で記述可能なマクロは必ず式の単位ごとにあらかじめ定められた括弧やコンマなどで区切ったものに限られる．一方，本研究のマクロシステムでは，ユーザが定義した区切り記号を利用することができ，また，`[# #]` 形式を利用した構造化によって，マクロの使用から無駄な符号を除去することができる．このような自由度を提供する目的から，本研究ではマクロパターンに応じた構文解析器の拡張というより洗練された実装技術を採用している．

本システムと同様に JavaScript 向けの Hygienic 構文マクロシステムに Mozilla の sweet.js[*5]がある．sweet.js も Scheme 風のパターンマッチ形式のマクロで，省略記号 ... による部分パターンの繰返しやメタ記号を用いた部分パターンのグループ化などの機能をサポートしている．sweet.js については実装の詳細は不明であり，明確な仕様書もないが，現時点での実装においては，ネスト構造を持ったパターンを含むような比較的複雑なマクロの取扱いに失敗する傾向にある．Arai らが実装した再帰下降式演算子順位文法を応用したマクロシステムと同様に，処理性能は高かったものの，マクロシステムの仕様の形式化ができなかった．それを踏まえて，本研究では展開性能よりも，記述の自由度と系統的に実現できることを重視した．

Bachrach らはマクロシステムをその内部形式に応じて，文字マクロシステム，字句マクロシステム，構文木マクロシステム，そしてスケルトンマクロシステムに分類している [3]．このなかで，構文木マクロはプログラムの抽象構文木に対する明示的な操作を許すための API を提供する方式である．たとえば，OCaml のための構文マクロシステムである Camlp5 は，OCaml の抽象構文木に対して 42 個の型を提供し，さらにそのうちの式と文はそれぞれ数十のデータ構成子からなる抽象データ型である [7]．このため Camlp5 を利用するためには，これらのデータ型の利用方法を学ぶ必要があり，マクロプログラマにとっては大きな負担となる．一方，スケルトンマクロシステムは式や文といった非常に少数の構文要素のみをプログラミング言語の構文レベルで提供する方式である．実際，スケルトンマクロシステムに分類される本提案が提供する構文要素は基本的には識別子（identifier），式（expression），文（statement）のみであり，さらにマクロ展開方式についてはホスト言語の文法に沿った簡単なパターンマッチで定義できるためマクロプログラマの負担は構文木マクロシステムに比べて大幅に軽減される．

Haskell のためのメタプログラミング用ライブラリである Template Haskell [19] は，構文木マクロとスケルトンマクロの両方の性質を持つ構文マクロシステムである．マクロ定義において展開後の構文木を表す手法が 3 種類あり，そのうち抽象構文木の型を表す代数的データ型を用いる手法と構文木の生成を補助する構文生成関数を用いる手法が構文木マクロに該当する．そして，Haskell の具象構文を用いて定義できる quasi-quote 記法を用いる手法がスケルトンマクロに該当する．これらの手法は，定義するマクロに応じて使い分けることができるが，複雑なものを定義するためには代数的データ型や構文生成関数を用いなければならず，こちらもその利用方法を学ぶ必要がある．Template Haskell において，マクロはコンパイル時に実行，すなわち展開される関数である．マクロ構文はすべて関数と同じ形式をしており，マクロを使用するときにはプログラム実行時に実行される関数と区別するための注釈を付けなければならない．一方で，本研究のマクロシステムはマクロ名でマクロの使用を判別するため，関数適用と同じ形のマクロ構文であっても，他の関数と同じように記述することが可能である．

関数型，オブジェクト指向，命令型言語の特徴を持つ静的型付け言語 Nemerle のマクロシステム [20] も Hygienic

[*5] http://sweetjs.org/

構文マクロシステムである．Nemerle も quasi-quote 記法を用いたマクロ定義を提供する．Nemerle で定義できるマクロは基本的には関数適用と同じ形に制限されており，それと異なる形のマクロを定義するためには，ユーザがマクロ構文を構成するトークンの並びを追加で記述しなければならない．一方，本研究のマクロシステムはマクロ構文の形式に関してマクロ名から始まること以外に大きな制限は設けておらず，また，パターンに直接マクロの入力形式を記述できるため，自由度の高いマクロ構文を簡潔に定義することができる．機能面においては，Nemerle では省略記号による部分パターンの繰返しや再帰的なマクロの定義ができない．一方，本研究のマクロシステムは Scheme のマクロと同じように省略記号 ... による部分パターンの繰返しをサポートしており，また，再帰的なマクロの定義も接尾辞パターンを導入したことにより，多くの場合において可能である．

10. まとめ

本稿では，JavaScript の構文拡張を可能にする Hygienic 構文マクロシステム EX-JS の実装について述べた．非 Lisp 言語において Hygienic 構文マクロを実現するには，拡張可能なパーザの実現と Hygienic 構文マクロ展開器の実装の 2 つの課題があったが，これらを解析表現文法と Scheme マクロ展開器の利用により解決した実装手法を提案した．マクロ定義からマクロ構文の文法記述を動的に生成し，それを基にパーザを再構成することで拡張可能なパーザを実現し，さらに，S 式との相互変換と Scheme マクロ展開器によるマクロ展開によって JavaScript でのマクロ展開を実現した．

今後の課題としては，まずパーザ再構成にかかる時間の改善があげられる．多大なパーザ生成時間の原因となる文法記述方法について調査し，その点を改良することで，より短い時間でのマクロ展開が可能になると考えている．また，8.3 節で述べたようにマクロが生む曖昧性の調査も今後の課題である．さらに，本実装方式を用いて JavaScript 以外の言語に Hygienic 構文マクロシステムを実装することも今後の課題である．

謝辞　本研究は JSPS 科研費 23500034 および 23700043 の助成を受けたものです．

参考文献

[1] Adams, IV, N.I., Bartley, D.H., Brooks, G., Dybvig, R.K., Friedman, D.P., Halstead, R., Hanson, C., Haynes, C.T., Kohlbecker, E., Oxley, D., Pitman, K.M., Rozas, G.J., Steele Jr., G.L., Sussman, G.J., Wand, M. and Abelson, H.: Revised^5 report on the algorithmic language Scheme, *SIGPLAN Not.*, Vol.33, pp.26–76 (1998).

[2] Arai, H. and Wakita, K.: An implementation of a hygienic syntactic macro system for JavaScript: A preliminary report, *Workshop on Self-Sustaining Systems*, *S3 '10*, New York, NY, USA, pp.30–40, ACM (2010).

[3] Bachrach, J. and Playford, K.: D-Expressions: Lisp Power, Dylan Style, Technical report (1999).

[4] Bawden, A. and Rees, J.: Syntactic closures, *Proc. 1988 ACM conference on LISP and functional programming*, *LFP '88*, New York, NY, USA, pp.86–95, ACM (1988).

[5] Clinger, W. and Rees, J.: Macros that work, *Proc. 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, *POPL '91*, New York, NY, USA, pp.155–162, ACM (1991).

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

[7] de Rauglaudre, D.: *Camlp5 — Reference Manual*, Institut National de Recherche en Informatique et Automatique, version 6.00 (2010).

[8] Ford, B.: Packrat parsing: simple, powerful, lazy, linear time, *Proc. 7th ACM SIGPLAN international conference on functional programming*, *ICFP '02*, New York, NY, USA, pp.36–47, ACM (2002).

[9] Ford, B.: Parsing expression grammars: A recognition-based syntactic foundation, *Proc. 31st annual ACM SIGPLAN — SIGACT symposium on principles of programming languages*, New York, NY, USA, pp.111–122, ACM (2004).

[10] Ghuloum, A. and Dybvig, K.: psyntax (online), available from ⟨http://www.cs.indiana.edu/chezscheme/r6rs-libraries/⟩.

[11] Jaffer, A.: *SLIB The Portable Scheme Library*, version 3b3 (2010) (online), available from ⟨http://people.csail.mit.edu/jaffer/slib.pdf⟩.

[12] Kernighan, B.W. and Ritchie, D.M.: *The C Programming Language*, Second Edition, Prentice Hall, Inc. (1988).

[13] Knuth, D.E.: *The TeXbook*, Addison-Wesley Professional (1984).

[14] Kohlbecker, E., Friedman, D.P., Felleisen, M. and Duba, B.: Hygienic macro expansion, *Proc. 1986 ACM conference on LISP and functional programming*, *LFP '86*, New York, NY, USA, pp.151–161, ACM (1986).

[15] Pratap, L. and Allen, W.-B. (Eds.): *Standard ECMA-262 ECMAScript Language Specification*, 5th edition, ECMA International (2009).

[16] Pratt, V.R.: Top down operator precedence, *Proc. 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, *POPL '73*, New York, NY, USA, pp.41–51, ACM (1973).

[17] Seindal, R., Pinard, F., Vaughan, G.V. and Blake, E.: *GNU m4 manual*, 1.4.16 edition (2011) (online), available from ⟨http://www.gnu.org/software/m4/manual/m4.pdf⟩.

[18] Shalit, A.: *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).

[19] Sheard, T. and Jones, S.P.: Template meta-programming for Haskell, *SIGPLAN Not.*, Vol.37, No.12, pp.60–75 (2002).

[20] Skalski, K., Moskal, M. and Olszta, P.: Meta-programming in Nemerle (2004) (online), available from ⟨http://nemerle.org/metaprogramming.pdf⟩.

[21] Sperber, M., Dybvig, R.k., Flatt, M., Van straaten, A., Findler, R. and Matthews, J.: Revised^6 report on the algorithmic language Scheme, *Journal of Functional Programming*, Vol.19, pp.1–301 (2009).

[22] Steele Jr., G.L.: *Common Lisp the Language*, 2nd edition, Digital Press (1990).

[23] Waddell, O. and Dybvig, R.K.: Extending the scope of syntactic abstraction, *Proc. 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, *POPL '99*, New York, NY, USA, pp.203–215, ACM (1999).

甫水 佳奈子　（正会員）

1988 年生．2011 年東京工業大学理学部情報科学科卒業．2013 年同大学大学院情報理工学研究科数理・計算科学専攻修士課程修了．同年，三菱電機株式会社入社．ソフトウェア開発に興味を持つ．

脇田 建　（正会員）

1989 年東京大学理学部情報科学科卒業．1991 年同大学大学院理学系研究科情報科学専攻修士課程修了．1992 年同大学大学院理学系研究科情報科学専攻博士課程を中退し，東京工業大学理学部情報科学科助手等を経て，現在，東京工業大学大学院情報理工学研究科准教授．博士（理学）．プログラミング言語の設計と実装，大規模社会ネットワーク解析，ユーザインタフェース等に興味を持つ．日本ソフトウェア科学会，ACM，IEEE 各会員．

佐々木 晃

1994 年東京工業大学理学部情報科学科卒業．2001 年同大学大学院情報理工学研究科博士後期課程単位取得退学，東京大学医科学研究所産学官連携研究員等を経て，法政大学情報科学部准教授．博士（理学）．エージェントシミュレーションとその処理系，プログラミング言語等の研究に従事．日本ソフトウェア科学会，ACM 各会員．