Hygienic構文マクロシステムを用いたJavaScriptプログラミング

脇田 建[1,a)]　甫水 佳奈子[1,b)]　佐々木 晃[2,c)]

概要：現在開発を進めている JavaScript 向けのマクロシステムの特徴は抽象構文木を変換対象とする構文マクロシステムであり，さらに文脈内におけるマクロ使用によって変数の参照関係が破壊されない Hygienic 性と呼ばれるある種の健全性を担保する点にある．本発表では，この Hygienic 構文マクロの表現力を披露する．

1. Hygienic マクロシステム

マクロはプログラミング言語開発の初期から存在し，今日に至るまで，常にプログラマにとって身近な存在であった．多くの人々が親しんでいる cpp や m4，あるいは TEX での命令記述に用いられている素朴なマクロシステムは**字句マクロ**と呼ばれ，字面の置き換えを行うのみである．このため，マクロプログラマはしばしば思いがけないマクロ展開結果と格闘することとなる．

一方，1970-1980 年代に LISP 文化のなかで育まれたより高度なマクロシステムが**構文マクロ**である．構文マクロは抽象構文木の変換系であるため，字句マクロのような問題を避けつつ，Common Lisp のオブジェクトシステムにあたる CLOS に代表されるようなかなり複雑な構文を定義することができる．

LISP の構文マクロシステムは 1980 年代後半に提案された変数参照に関する健全性 (Hygienic 性) とその効率的な実装技術の研究によって 1990 年代に一応の完成を見た．[2], [3], [5], [6]

Hygienic マクロシステムはその有用性は認められているものの，それを LISP 以外の言語に実装した例はほとんど見当らない．LISP において，マクロシステムの研究が進んだ理由はその括弧を多用するメタ構文に負うところが多い．一方，一般のプログラミング言語は，系統的に定まったメタ構文はないため，多くのプログラマにとってとっつきやすい面はあるものの，構文マクロの構成は困難であった．

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
2 法政大学情報科学部
a) wakita@is.titech.ac.jp
b) homizu8@is.titech.ac.jp
c) asasaki@hosei.ac.jp

われわれは，Hygienic 構文マクロを一般のプログラミング言語に導入するための系統的な実装手法について研究を行い，現在，JavaScript を母言語としたマクロシステムの研究を行っている．[1], [4], **?**本発表においては，現在実装が進んでいるプロトタイプを利用して記述したマクロを紹介したい．

2. Hygienic マクロプログラミング

以下では，前節で概説した Hygienic マクロを用いて何ができるかを例題を通して示しつつ，既存の単純なマクロシステムとの違い，強力な記述力，安全性などを紹介する．

2.1 単純なマクロの例 (add)

本システムは，JavaScript の式と文の構文を拡張する機能を提供している．式と文を定義するマクロはそれぞれ **expression** 宣言，**statement** 宣言を用いる．

マクロプログラミングの最初の例として，数値の加算に対応した **add** マクロを見てみよう．

```
expression add {
  expression: e1, e2;
  { add e1  e2 => e1 + e2 }
}
console.log(add 1 2);
console.log((add 1 2) * 5);
```

この例の最初の 4 行がマクロの定義である．最初の行は **add** というマクロ名を宣言している．本システムのマクロは，マクロの使用を識別子で判断するために，必ずマクロ名で始まる．このため，識別子で始まらない構文，あるいは新しい演算子の定義などはできない．

3 行目はマクロの構文と対応する JavaScript の構文を定

義している．ここで，**add** が二つの引数を空白で区切って与えること，そしてマクロ展開によって `e1+e2` という JavaScript の式に変換されることが定義されている．

このように，本システムではマクロの定義はパターンマッチによって与える．世の中の多くの構文マクロシステムは，プログラマに構文解析木への API を提供して，構文木上の変換器を実装することを許しているが，その利用のためには構文解析木について学ばなくてはならず，さらに構文解析木を操作するための能力を要する．一方，本システムはソースコードレベルでのパターンマッチ規則を定義するだけでマクロが定義できるためプログラマの負担は大幅に軽減される．パターンマッチ規則から，変換器を生成する部分はシステムが完全に面倒をみる設計となっている．

本システムでは，プログラマが与える変換規則を解析し，JavaScript の構文解析器に新しい構文を追加し，さらに変換規則から変換器を自動生成する．特に構文解析器の生成のためには，プログラマが変換規則の記述において利用するマクロ変数がどのような構文要素に該当するかという知識が必要となる．この情報を与えるために，本システムではマクロ宣言のなかでマクロ変数に対応する構文要素の種類を宣言することとしている．

上述の **add** マクロに出現する二つのマクロ変数 $e1$ と $e2$ はどちらも式に相当するため，それらは **expression** と宣言されている．後述の例に出てくるが，このほかに指定できる構文要素には文に相当する **statement** と識別子に相当する **identifier** などがある．

上述の例の 5-6 行目において，いま定義した **add** マクロを使用している．このマクロは **expression** 宣言によって式として宣言されたため，JavaScript の式が許される任意の箇所で利用することができる．

本システムはマクロの宣言と使用を含む JavaScript のプログラムを受け取り，マクロ展開を施したマクロが出現しない JavaScript のプログラムを生成する．以下が前述の例を展開した例である．

```
(console.log ((1 + 2)));
(console.log (((1 + 2) * 5)));
```

2.2 文マクロを利用した新しい構文の定義 (unless)

次に文を定義するための **statement** 宣言を利用した例を見よう．ここでは，与えられた条件が満たされない場合にのみ式を実行するための **unless** 構文を定義する．

マクロ定義の方法は **statement** 宣言を用いることを除けば **add** マクロと同様である．ほとんど同じ単純な宣言方法によって式も文も定義できることは，他の多くの構文マクロシステムと比較して，本システムの大きな特徴である．

```
statement unless {
  expression: C;
  statement: S;
  { unless (C) S => if (!C) S }
}
function test(t, c1, c2) {
  if (c1)
    unless (c2) console.log(t, 'P1');
  else console.log(t, 'P2');
}
test(1, true, true);        // 出力なし
test(2, true, false);       // P1
test(3, false, undefined); // P2
```

一般的に用いられている `cpp` や `m4` のような字句マクロの場合，マクロ展開は字面に対してなされるため，マクロ定義やマクロの使用にあたっては特別な注意を要する．一方，構文マクロでは，マクロ展開は抽象構文木に対してなされるために，マクロが使用されている外側の文脈を破壊することなくマクロ展開がなされる．

たとえば，上の **unless** マクロを字句マクロを用いて展開した場合，本来，**unless** マクロの外側の文脈にある **else** 文が **unless** マクロが導入する **if** 文と対応し，構文の構造が変化する恐れがある．

本システムは構文マクロシステムであるので，そのような心配は無用である．以下のマクロ展開例で見られるように，字句マクロシステムにおけるような問題が発生していないことがわかる．なお，プログラム例で各 *test* 関数の呼び出しにその出力結果をコメントとして添えた．

```
if (c1_192_) {
  if (!c2_192_) { console.log(t_192_, "P1"); }
} else { console.log(t_192_, "P2"); }
```

2.3 変数束縛に関連した Hygienic 性 (let)

これまでに **add** マクロと **unless** マクロの例を通して，式と文の構文を定義する方法について述べた．これまでの例は，マクロの使用方法を説明するための例であって，必ずしも有用なマクロとは言えない．次に紹介する **let** は JavaScript に望まれた機能のひとつであり，次世代の標準に追加される見通しの機能のひとつである．

JavaScript のスコープ規則によれば，関数内で宣言された局所変数のスコープはその関数の本体全体ということになっている．これは，関数の内側に波括弧を用いた場合も，**for** 文のなかでループ変数を宣言した場合もである．この不便を避けるために，しばしば無名関数を用いることがあるが，構文的に複雑であり，プログラムの可読性が低減する．

以下では，式 $e2$ の評価において，式 $e1$ の評価値を変数 v に束縛するための **let** マクロを定義している．ここで **let** マクロは前述の無名関数を用いた JavaScript のコードに変換するものである．式 $e2$ を評価する無名関数の仮引数 v

に対する実引数として評価値を束縛したい式 (*e1*) を与えることで，局所スコープが構成されている．

```
expression let {
  identifier: v;
  expression: e1, e2;
  keyword: In;
  { let v = e1 In e2 =>
    (function (v) { return e2; })(e1) }
}
var where = 'global';
let where = 'local'
In console.log(where); // local
console.log(where);    // global
```

マクロの使用例では，同名の二つの変数 *where* を用いているが，**let** マクロが構成する局所環境で宣言された値と，大域環境において宣言された値がそれぞれ適切に参照できることが示されている．

構文マクロは，マクロが使用される文脈の構造を破壊しないことを保証するものの，その文脈の意味構造を保証するものではない．すなわち，マクロ使用の文脈で定義された変数と同名のものがマクロ展開によって導入された場合には，名前が衝突する可能性がある．この問題を避けるために LISP のマクロプログラミングでは新鮮な名前を生成する関数を利用することが求められる．

一方, 本システムは構文マクロであるとともに, Hygienic 性も併せて提供している．Hygienic 性は，前述のような思いがけない名前の衝突が起こらないことを保証する性質である．Hygienic 性を保証するためにマクロ展開器は局所変数に α 変換を施す．たとえば，上のコードは以下のように展開される．ここで，大域変数の名前は変化していないが，局所変数は *where_193_* に変更されている．

```
var where = "global";
(function (where_193_) {
  return (console.log (where_193_));
} ("local"));
(console.log (where));
```

このマクロの定義において，**identifier** 宣言子と **keyword** 宣言子を用いている．それぞれ，識別子と予約語に相当する構文要素にあたる．

この例では **let** マクロの **In** 節のなかに一文しか使われていないが，波括弧を用いた重文を用いれば **In** 節のなかで複数の文を記述することもできる．

2.4 列とシンボルの扱い (Path)

秘書さんに地元から箱根に移動するための経路情報を記述してもらうために，以下のような日本語っぽい記法を使うこととしてみよう．実は，いかにも日本語のような表現は，本システムで記述したマクロであり，これをそのまま実行することもできる．

```
console.log(
  乗り換え経路
    大岡山 から 武蔵小杉 まで 東急目黒線 で
    武蔵小杉 から 横浜 まで 東急東横線 で
    横浜 から 小田原 まで 東海道本線 で
    小田原 から 強羅 まで 箱根登山線 で 行きます);
```

このような記述を可能にする乗り換え経路マクロの定義は以下の通りである．まず，このマクロは複数の乗り継ぎが記述できる点に特徴がある．このために本システムは，繰返し構造を記述するための [# #] ... 記法を提供している．これは [# #] で囲われたパターンの繰返しを意味している．つまり，以下の定義では，乗り換え経路マクロの記述のなかで，「〜 から 〜 まで 〜 で」という記述が繰返されることを指示している．

本システムの元となっている Scheme の Hygienic マクロにも ... パターンを利用できるが，Scheme の場合は繰返される構造を ( ) で囲うのがいかにも LISP としては自然な表現に思える．本システムが想定する LISP ではない言語においては任意の式，任意の文を覆うための適切なブロック構造はないこと，さらに，マクロの使用において括弧を用いない構造化を許すことで，より自然な構文が記述できることを目指した．このために，パターンにおける繰返し構造を明記する目的で仮想的な括弧記法の [# #] 節を導入した．

この ... パターンはマクロの展開方法を記述したテンプレート部でも利用できる．本システムにおいては，マクロの文法を規定するパターン部の構造とテンプレート部の構造を見比べ，マクロ変数の対応関係を解析することで，適切にマクロ変換器を構成している．

乗り換え経路マクロの記述においてもう一点特徴的なのは，大岡山，武蔵小杉などの文字列データに文字列を表す引用符を用いていない点であろう．このような自由記述を許すために導入した機能が **symbol** である．本システムにおいて **symbol** 宣言されたマクロ変数は，構文解析時に識別子として記述を読み込んだものを内部的に文字列に変換して処理する．このため，たとえば *A* に合致する「大岡山」という識別子は"大岡山"という文字列に変換されて処理される．

```
expression 乗り換え経路 {
  symbol: A, B, 路線;
  keyword: から, まで, で行き, で, 行きます;
  { 乗り換え経路
    [# A から B まで 路線 で #] ...
    行きます =>
  [{ from: A, to: B, via: 路線 }, ...] }
}
```

このように記述したマクロを実行すると，以下のような

図 1　2.5 節のマクロ記述によって描画された図形

出力が得られる．マクロを利用することによって，プログラミング言語的な記述を毛嫌いする人々にもプログラミングに触れる機会が与えられることを期待したい．

```
[ { from:'大岡山', to:'武蔵小杉', via:'東急目黒線' },
  { from:'武蔵小杉', to:'横浜', via:'東急東横線' },
  { from:'横浜', to:'小田原', via:'東海道本線' },
  { from:'小田原', to:'強羅', via:'箱根登山線' } ]
```

2.5　おしまいに (SVG)

本稿最後のマクロの例では，ウェブブラウザの Smart Vector Graphics (SVG) を描画する API を利用したものである．この記述の最後にブラウザの画面に青い長方形と白地に赤い線で囲われた円を描くコードが含まれている．

このプログラムをマクロ展開したものを HTML に読み込むことでマクロを用いた JavaScript をウェブコンテンツの記述にも利用することができる．

```
expression let {
  identifier: v;
  expression: e1, e2;
  keyword: In;
  { let v = e1 In e2 =>
    (function (v) { return e2; })(e1) }
}

function SVG$Shape(type, plist) {
  var shape =
    document.createElementNS(
      "http://www.w3.org/2000/svg", type);
  plist.forEach(function (attr_value) {
      shape.setAttribute(
        attr_value[0].replace(/_/g, '-'),
        attr_value[1]);
    });
  return shape;
}

expression Rect {
  expression: value; symbol: attr;
  { Rect [# attr: value #] ... =>
    SVG$Shape('rect', [[attr, value], ...]) }
}

expression Circle {
  expression: value; symbol: attr;
  { Circle [# attr: value #] ... =>
    SVG$Shape('circle', [[attr, value], ...]) }
}

$(function () {
    var add =
      let svg = $('#svg').get()[0] In
        function (shape) {
          svg.appendChild(shape); };

    add(Rect x:20 y:20 width:100 height:80
      fill:'blue');
    add(Circle cx:200 cy:60 r:40
      fill:'white' stroke:'red' stroke_width:3);
  });
```

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] Hiroshi Arai and Ken Wakita. An implementation of a hygienic syntactic macro system for javascript: a preliminary report. In *Workshop on Self-Sustaining Systems, S3*, pp. 30–40. ACM, 2010.

[2] William Clinger and Jonathan Rees. Macros that work. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 155–162, New York, NY, USA, 1991. ACM.

[3] Chris Hanson. A syntactic closures macro facility. *SIGPLAN Lisp Pointers*, Vol. IV, No. 4, pp. 9–16, 1991.

[4] Kanako Homizu, Ken Wakita, and Akira Sasaki. A proposal of implementation technique for hygienic syntactic macro system for JavaScript. poster presentation at 10th Asian symposium on programming languages and systems (APLAS 2012), December 2012.

[5] Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *LFP '86: Proceedings of the 1986 ACM conference on LISP and functional programming*, pp. 151–161, New York, NY, USA, 1986. ACM.

[6] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. *Revised^6 report on the algorithmic language Scheme*, September 2007.

付　　録

A.1　デモでお答えします

○どうして add マクロを括弧に囲っているの／複数の変数を束縛する let は使えないの／使ってみたいんだけど／実装あるの／どうやって実装したの／...