アリスプの少女 ハイジニックマクロ

対話による Scheme Hygienic Macro 入門

stibear (@stibear1996)

はじめに

1　イントロダクション

Scheme というプログラミング言語をご存知でしょうか？ Scheme は，Lisp 方言のひとつで，Lisp 方言には他にも，Common Lisp や Emacs Lisp，Clojure などがあります．その中でも Scheme は，プログラミングに必要な言語機能を極限まで抽象化するという，最小主義的[1]な一面を持つ Lisp です．

Lisp の特徴として，そのシンタックスがありますが，この夥しいほどの括弧による表現は，Lisp に大きなパワーをもたらします．つまり，Lisp は括弧によって，プログラムとデータとが同じ形式であるという，同図像性（Homoiconicity）を得るのです．そして，この同図像性を用いて，Lisper 達は強力なマクロを生み出します．

しかし，このマクロも万能ではなく，弱点[2]がありました．それは，変数捕捉と呼ばれます．それは，マクロ展開が名前の衝突を起こした時に発生します．これは，ほんの些細なバグにつながることもあり，しかも，それは時としてエラーとして通知されません．

この弱点に対して，様々なアプローチが考えだされました．例えば Common Lisp では，伝統的なマクロシステムはそのままで，gensym というユニークな名前を生成する関数を用いて，名前の衝突を回避しています．しかし，Scheme にとってこの問題はより深刻なものでした．というのも，Common Lisp と異なり，Scheme は Lisp-1，つまり，変数や関数の名前空間が一つなのです[3]．Scheme は，Common Lisp のような手段を取らず，Hygienic Macro（健全なマクロ）というマクロシステムによって解決を試みます．

この Scheme の Hygienic Macro について，タイトルにある通り，少女とともに，学んでいきます．

---

[1] 実際，R^{5}RS（Scheme の仕様書）は 50 ページしかありませんでした．

[2] しかし，これも上手く利用することで，便利なものとなります．

[3] 対して Common Lisp のような名前空間が複数あるものは，Lisp-2 と呼ばれます．ただし，名前空間が 2 つ以上あろうと，Lisp-2 と呼ばれます．

2　今回のおはなしについて

あらすじ

今回は，アリスプと言うヨー○ッパのとある山岳地帯に住む，ハイジという少女が主人公です．

ハイジは，事情があって，アリスプに住む親戚のおじいさんに預けられ，一緒に暮らしています．そこで，Lisp エイリアンを始めとする，大自然に生きる動植物達と出会います．ハイジは，厳しくも優しく，懐の深さを感じさせてくれるこのアリスプの大自然をいつしか愛するようになり，環境を守ることの大切さを学んでいきます．

登場人物

- ハイジニック＝マクロ（ハイジ）
  アリスプに住むごく普通の少女．アリスプの自然が大好き．
- CLer ら（クララ）
  Common Lisp を信奉する，いたいけな少女（たち？）．
- ALGOL ムおんじ（アルムおんじ）
  ハイジと一緒にアリスプに住むおじいさん．ハイジの母親の Scheme とは親戚．
- ペーター＝ノーヴィグ（ペーター）
  アリスプに住む，Lisp エイリアン飼いの少年．AI の権威．

目次

第 I 章

ハイジニックなマクロって？

1　Lisp のマクロ

マクロってなに？

> ハイジ「おじいさん，マクロってなに？」
> おんじ「置き換えのことだよ」
> ハイジ「置き換え？」
> おんじ「そうだ．マクロによって，受け取った S 式を別の S 式に置き換えるんだ」
> ハイジ「どうしてそんなことできるの？」
> おんじ「Lisp の同図像性のおかげさ」
> ハイジ「ふーん」
>
> ―――――――――――――――――――――――――――――――
> アリスプの少女　ハイジニックマクロ　第 1 話

「はじめに」で述べた通り，Lisp には同図像性があります．Lisp では，データ構造としてのリストの表現と，プログラムとしての関数呼び出し（または，特殊フォーム）の表現は，同じシンタックスです．したがって，それを評価すると関数呼び出しになるようなリストを作り，評価させることによって，プログラムを書くプログラムを作ることができます．これが，Lisp のマクロです．

Lisp のマクロは，S 式を受け取り，それを別の S 式に変換して返す関数のようなものです．この変換をマクロの「展開」と呼びますが，マクロ展開はプログラムが解釈される前に行われます．

伝統的マクロ

> クララ「ハイジ，マクロはいいものよ．私もよく使うわ」
> おんじ「クララが使うのは伝統的マクロと呼ばれるものだな」
> ハイジ「？（首を傾げる）」
> おんじ「クララが使う Common Lisp では，defmacro を使ってマクロを書くんだ」
> ハイジ「それを伝統的マクロというの？」
> おんじ「そうだ．今までの色々な Lisp で使われてきたものと同じ仕組みだからな」
>
> アリスプの少女　ハイジニックマクロ　第 2 話

R^{5}RS では，いわゆる伝統的マクロシステムのようなマクロは規定されてませんが，処理系によっては独自に用意されていることもあります．ここでは，あとで述べる Scheme の健全なマクロシステムとの比較のために，Common Lisp などの `defmacro` と同様の，Gauche の伝統的マクロシステムである `define-macro` を基準に，伝統的マクロシステムについて解説していきます．

On Lisp における，`nil!`[1]を定義してみましょう．`(nil! x)` としたとき，`(set! x '())` と同じ効果が得られます．

```
(define-macro (nil! x)
  (list 'set! x ''()))
```

この定義を評価した以降，`(nil! var)` の形の式を見ると，`(set! var '())` と展開されるようになります．

伝統的マクロでは，マクロが生成した式は，その呼び出し元で評価されます．マクロ展開は，式の評価に先行して行われます．そのため，マクロを含む式は，あたかもはじめから，その展開後の式がそこにあったかのように評価されます．マクロ展開が式の評価に先んじて為されるということは，しっかり認識しておく必要があります．

ここでは，マクロがどのように使われ，Lisp プログラミングにおいて，どんな効果をもたらすのかを詳細に示すことはしません．Common Lisp においてのマクロの基礎については，昨年度の部誌に収録されている，拙著「マクロのおはなし[2]」をお読みください．

[1] nil というのは，Scheme における空リストであり，#f であると言えます．
[2] 灘高パソコン研究部 作品紹介 (http://www.npca.jp/works/)

マクロの弱点

> クララ「ね？ マクロはいいものでしょ？」
> ハイジ「ええ．便利だと思うわ！」
> おんじ「だが，万能ではないのだ」
>
> アリスプの少女　ハイジニックマクロ　第 3 話

例題として swap!マクロを考えます．引数を 2 つ取り，それぞれの束縛を入れ替えます．次のように使用します．

```
(let ((x 3) (y 5))
  (swap! x y)
  (list x y))
;=> (5 3)
```

swap!マクロは次のように定義できます．

```
(define-macro (swap! a b)
  `(let ((tmp ,a))
     (set! ,a ,b)
     (set! ,b tmp)))
```

ほとんどの場合，このマクロはちゃんと動くでしょう．しかし，tmp という名前の変数が引数に渡された場合，たちまちこのマクロは動かなくなってしまいます．

```
(let ((tmp 3) (y 5))
  (swap! tmp y)
  (list tmp y))
;=> (3 5)
```

Gauche には，macroexpand という関数があります．この関数によってマクロの展開結果を確認することができます．

```
(let ((tmp 3) (y 5))
  (macroexpand '(swap! tmp y)))
;=> (let ((tmp tmp))
;      (set! tmp y)
;      (set! y tmp))
```

このようなバグは，マクロ引数が捕捉されてしまっているために起こるものです．伝統的

マクロではこれを gensym というインターンされていない新しいシンボルを返す関数を用います．返されるシンボルは他のシンボルとは決して eq?にならないことが保証されているので，変数捕捉は起こりません．gensym を使った swap!マクロの定義は次の通りです．

```
(define-macro (swap! a b)
  (let ((tmp (gensym)))
    `(let ((,tmp ,a))
       (set! ,a ,b)
       (set! ,b ,tmp))))
```

このように，伝統的マクロでは，gensym のような関数によって変数捕捉を回避しています．しかし，マクロシステムに伝統的マクロを採用している Common Lisp とは異なり，Scheme は，変数と関数などの名前空間がすべて同一です．そのため，すべてのシンボルに gensym を呼ぶのは冗長で，効果的な手段とは言えません．そこで，プログラマの手を煩わせることなく，自動的に名前衝突を回避することが重要視されるようになり，健全性という考え方が生まれました．

2　健全なマクロ

syntax-rules

> ハイジ「伝統的マクロがどんなのかってことは分かったわ．じゃあ Scheme には仕様で規定されたマクロはないの？」
> おんじ「Scheme でのマクロかい？ syntax-rules が R^{5}RS 以降では規定されているぞ」
> ハイジ「syntax-rules？」
> おんじ「ああ，そうだ．だが，少し癖があるがな」
>
> ―― アリスプの少女　ハイジニックマクロ　第 4 話

Scheme でマクロを書くには，複数の方法がありますが，R^{5}RS に準拠した Scheme では，syntax-rules を使います．例えば，次に示すような，条件が満たされるとき，複数の式が評価されるマクロ when を定義したいとします．

```
(when (= a 10) (display a) (+ a 20))
```

これを syntax-rules で書くと，次のようになります．

```
(define-syntax when
  (syntax-rules ()
    ((_ pred body ...)
     (if pred (begin body ...)))))
```

ここで，define-syntax は，マクロを定義するための特殊オペレータです．構文は次の通りです．

```
(define-syntax <keyword> <transformer spec>)
```

キーワードは定義するマクロの名前，式は，評価された結果がマクロトランスフォーマである必要があります[3]．

このように，syntax-rules では，パターン言語を使って，受け取った S 式とパターンマッチをします．そして，マッチしたパターンを指定されたテンプレートに沿って展開します．syntax-rules の構文は次の通りです．

```
(syntax-rules (<literal> ...)
  (<pattern> <template>) ...)
(syntax-rules <ellipses> (<literal> ...)
  (<pattern> <template>) ...)
```

リテラル部には，マクロ中で使われる識別子を列挙したリストを与えます．そして，それ以降に構文規則を書いていきます．

最終的に，syntax-rules のインスタンスは，健全な書き換え規則の列を規定することで，新しいマクロトランスフォーマを生成します．

健全性と水準

> クララ「syntax-rules は健全だっていうけど，パターン言語は気持ち悪いと思うわ．defmacro でいいじゃないの」
> ハイジ「おじいさん，syntax-rules と defmacro では，どう違うの？」
> おんじ「syntax-rules は健全で高水準なんだ．defmacro は健全でなく，低水準なマクロシステムだ」
> ハイジ「けんぜん？こうすいじゅん？」
>
> アリスプの少女　ハイジニックマクロ　第 5 話

マクロシステムには，健全であるかそうでないか，高水準か低水準かという 2 つの角度から見ることができます．おじいさんの言うとおり，Scheme の syntax-rules は，健全かつ高水準なマクロシステム，Common Lisp の defmacro は，健全でなく，低水準なマクロシステムと言えます．

既に述べた通り，Scheme は Lisp-1 であるため，名前の衝突は非常に大きな問題です．そのため，これを自動的に回避するために健全性という概念が生まれました．これによって，Scheme では，関数だけでなく，マクロもレキシカルスコープの概念によって説明する

[3] マクロトランスフォーマについては後述．

ことが可能になりました．

ところで，Scheme の標準マクロシステムは，R[6]RS で syntax-case が追加され，syntax-rules と syntax-case の 2 つになりました[4]．syntax-case では，変数捕捉をするマクロを書くことができます[5]．例えば，aif[6]の定義は次のようになります．

```
(define-syntax (aif stx)
  (syntax-case stx ()
    ((aif expr then else)
     (with-syntax ((it (datum->syntax #'aif 'it)))
       #'(let ((it expr))
           (if it then else))))))
```

これら 2 つが Scheme の標準のマクロシステムですが，処理系によっては，他にもマクロシステムが用意されていることがあります．defmacro のような，所謂伝統的マクロと呼ばれるマクロシステムが用意されていることもありますが，他にも，健全なマクロシステムでは次の 4 つのうちのどれか（若しくは全て）が用意してあるようです．

- Syntactic Closure
- Reversed Syntactic Closure
- Explicit Renaming
- Implicit Renaming

あとで述べるとおり，これらは全て，健全で低水準なマクロシステムです．

本記事では，これら低水準ハイジニックマクロについても解説していきますが，これらのマクロシステムが利用できる主な Scheme 処理系は次のとおりです．

Chibi-Scheme
: Syntactic Closure，Reversed Syntactic Closure，Explicit Renaming が使用可能．

CHICKEN Scheme
: Syntactic Closure，Reversed Syntactic Closure，Explicit Renaming，Implicit Renaming が使用可能．

MIT/GNU Scheme
: Syntactic Closure，Reversed Syntactic Closure，Explicit Renaming が使用可能．

[4] Common Lisp でいうところの，シンボルマクロのような identifier-syntax というものもあります．
[5] syntax-rules でも書けないことはありません．
[6] (aif (member 'b '(a b c)) (car it)) のように使います．

Picrin

Syntactic Closure，Reversed Syntactic Closure，Explicit Renaming，Implicit Renaming が使用可能.

Sagittarius Scheme

Explicit Renaming が使用可能.

第 II 章

syntax-rules と syntax-case

1　syntax-rules

パターン言語によるマクロ

> ハイジ「ギンギラギンにさり気なく〜♪」
> クララ「どうしたの？急に歌い出して」
> ハイジ「宇治金時食べたいわ！」
> クララ「近藤真彦（マッチ）に抹茶（マッチャ）かき氷ということね…」
> クララ「犬の卒倒（ワンパターン）だわ…」
> おんじ（ネタが古いな…）
>
> アリスプの少女　ハイジニックマクロ　第 6 話

syntax-rules は，パターン言語を用いた高水準なマクロであることは，I 章で述べました．ここでの高水準というのは，パターン言語が，Scheme の実行とは全く隔離された，独立したものであるということです．例えば，C 言語におけるプリプロセッサは，C 言語そのものとは全く独立した高水準なマクロシステムと言えます．

syntax-rules の構文は次の通りです（再掲）．

```
(syntax-rules (<literal> ...)
  (<pattern> <template>) ...)
(syntax-rules <ellipses> (<literal> ...)
  (<pattern> <template>) ...)
```

上から順に，<pattern>とマッチさせていき，マッチした場合，対応する<template>と置き換えられます．マクロ内で使われる予約語に関しては，<literal>として予め列挙しておきます．

高水準マクロの落とし穴

> クララ「ねえ，このマクロ，うまく展開されないんだけど，どうしてかしら？」
> おんじ「ああ，マクロの中で別のマクロを呼んでいるんだな？」
> おんじ「そういう時は…」
>
> ——アリスプの少女　ハイジニックマクロ　第 7 話

わかりやすさのために，実際のコードを見て議論していきます．

```
(define-syntax append-macro
  (syntax-rules ()
    ((_ (x ...) (y ...))
     '(x ... y ...))))

(define-syntax reverse-macro
  (syntax-rules ()
    ((_ (obj ...))
     (reverse-macro% (obj ...) ()))))

(define-syntax reverse-macro%
  (syntax-rules ()
    ((_ () x) 'x)
    ((_ (y1 y2 ...) (x ...))
     (reverse-macro% (y2 ...) (y1 x ...)))))

(define-syntax append-reverse-macro
  (syntax-rules ()
    ((_ (x ...) (y ...))
     (append-macro (reverse-macro (x ...)) (y ...)))))

(append-reverse-macro (1 2 3) (4 5 6))
; => (reverse-macro (1 2 3) 4 5 6)
; expected: (3 2 1 4 5 6)
```

ここでは，append-reverse-macro が expected 以下のように展開されることを期待しますが，実際は get-vars で始まる式で展開が止まってしまい，エラーとなります．これは，syntax-rules によるマクロが評価戦略が非正格評価（正規順序，もしくは名前呼び）で

あるため，一番外の式から展開されるからです．

これを解決するために，マクロ定義の CPS 変換を用います．なぜ CPS 変換を用いるのかの理由はここでは詳しく述べません[1]が，直感的にわかるのではないかと思います．

```
(define-syntax append-macro
  (syntax-rules ()
    ((_ (x ...) (y ...))
     '(x ... y ...))))

(define-syntax reverse-macro-cps
  (syntax-rules (syntax-lambda)
    ((_ (syntax-lambda cont-args cont-body) (obj ...))
     (reverse-macro-cps%
      (syntax-lambda it
        (let-syntax
            ((cont-syntax
              (syntax-rules ()
                ((_ cont-args) cont-body))))
          (cont-syntax it)))
      (obj ...) ()))))

(define-syntax reverse-macro-cps%
  (syntax-rules (syntax-lambda)
    ((_ (syntax-lambda cont-args cont-body) () x)
     (let-syntax
         ((cont-syntax
           (syntax-rules ()
             ((_ cont-args) cont-body))))
       (cont-syntax x)))
    ((_ (syntax-lambda cont-args cont-body) (y1 y2 ...) (x ...))
     (reverse-macro-cps%
      (syntax-lambda it
        (let-syntax
            ((cont-syntax
              (syntax-rules ()
```

[1] CPS 変換によって，正格評価と非正格評価を行き来することができます．

```
                    ((_ cont-args) cont-body))))
               (cont-syntax it)))
          (y2 ...) (y1 x ...)))))

(define-syntax append-reverse-macro-cps
  (syntax-rules (syntax-lambda)
    ((_ (syntax-lambda cont-args cont-body) (x ...) (y ...))
     (reverse-macro-cps
      (syntax-lambda it
        (let-syntax
            ((cont-syntax
              (syntax-rules ()
                ((_ cont-args) cont-body))))
          (cont-syntax ((append-macro it (y ...))))))
      (x ...)))))

(define-syntax append-reverse-macro
  (syntax-rules ()
    ((_ (x ...) (y ...))
     (append-reverse-macro-cps (syntax-lambda (it) it) (x ...) (y ...)))))
```

CPS 変換された append-reverse-macro はこのようになります．若干複雑なので，マクロ展開を順に追ってみましょう．（syntax-rules によって挿入される it シンボルは，それぞれわかりやすいように rename してあります．）

```
(append-reverse-macro-cps (syntax-lambda (it0) it0) (1 2 3) (4 5 6))

(reverse-macro-cps
 (syntax-lambda it1
   (let-syntax
       ((cont-syntax
         (syntax-rules ()
           ((_ (it0)) it0))))
     (cont-syntax ((append-macro it1 (4 5 6))))))
 (1 2 3))

(reverse-macro-cps
 (syntax-lambda it2
```

```
  (let-syntax
      ((cont-syntax
        (syntax-rules ()
          ((_ it1)
           (let-syntax
               ((cont-syntax
                 (syntax-rules ()
                   ((_ (it0)) it0))))
             (cont-syntax ((append-macro it1 (4 5 6)))))))))
    (cont-syntax it2)))
 (1 2 3) ())

(let-syntax
    ((cont-syntax
      (syntax-rules ()
        ((_ it5)
         (let-syntax
             ((cont-syntax
               (syntax-rules ()
                 ((_ it4)
                  (let-syntax
                      ((cont-syntax
                        (syntax-rules ()
                          ((_ it3)
                           (let-syntax
                               ((cont-syntax
                                 (syntax-rules ()
                                   ((_ it2)
                                    (let-syntax
                                        ((cont-syntax
                                          (syntax-rules ()
                                            ((_ it1)
                                             (let-syntax
                                                 ((cont-syntax
                                                   (syntax-rules ()
                                                     ((_ (it0)) it0))))
                                               (cont-syntax
```

```
                                                    ((append-macro
                                                      it1 (4 5 6))))))))))
                                                (cont-syntax it2))))))
                                            (cont-syntax it3))))))
                                        (cont-syntax it4))))))
                                    (cont-syntax it5))))))
                            (cont-syntax (3 2 1)))
```

CPS 変換は，機械的な変換なので，丁寧に順々と追っていくことで理解できるはずです．

2 syntax-case

低レベルマクロ

> ハイジ「おじいさん，aif マクロを作りたいわ！」
> おんじ「そうか，ならこの syntax-case を使ってみるといい」
> ハイジ「うん！でも，難しそうね」
>
> アリスプの少女　ハイジニックマクロ　第 8 話

いきなりですが，syntax-case の構文を示しておきます．

```
(syntax-case <expression> (<literal> ...)
  <clause> ...)

<clause> := (<pattern> <output-expression>)
    | (<pattern> <fender> <output-expression>)
```

<pattern>の識別子は，syntax-rules 同様，下線（_）やリテラルリストに出現する識別子，省略記号（...）以外はすべてパターン変数として解釈されます．<fender>式は，<pattern>にマッチした場合の追加条件となります．

syntax-case では，パターン変数は syntax 式内でのみ参照可能です．構文糖衣として #' も用意されています．

ここからは，aif の実装を見ながら解説していきましょう．syntax-case を用いた aif の実装は，次のとおりです．

```
(define-syntax aif
  (lambda (expr)
    (syntax-case expr ()
      ((aif pred then)
       #'(aif pred then #f))
```

```
    ((k pred then else)
     (with-syntax ((it (datum->syntax #'k 'it)))
       #'(let ((it pred))
           (if it then else)))))))
```

ちなみに，with-syntax の代わりに quasisyntax を利用することもできます[2].

```
(define-syntax aif
  (lambda (expr)
    (syntax-case expr ()
      ((aif pred then)
       #'(aif pred then #f))
      ((k pred then else)
       (let ((it (datum->syntax #'k 'it)))
         #`(let ((#,it pred))
             (if #,it then else)))))))
```

まず，低レベルマクロの要となるのが，datum->syntax 手続きです．datum->syntax はテンプレート識別子と任意の値を引数に取り，そのテンプレート識別子と同じ文脈情報をもつ，第 2 引数の値の構文オブジェクト表現を返します．

with-syntax は，let のようにしてパターン変数を束縛します．また，quasisyntax は，quasiquote と同様に用いられます．unquote，unquote-splicing と対応する，unsyntax，unsyntax-splicing が存在します．

R^{6}RS でのマクロの扱い

> ペーター「ふーん，R^{6}RS では，R^{5}RS に比べてマクロの扱いが変わったね」
> ペーター「syntax-case のような低レベルマクロを追加したわけだし，しょうがないか」
> クララ（何を独りでぼやいてるのかしら…）
>
> アリスプの少女　ハイジニックマクロ　第 9 話

R^{6}RS では，低レベルマクロが追加されたため，マクロシステム周辺で幾つかの大きな変更がありました．syntax-case は，健全性を保ったまま，Scheme 自身を使ってマクロを書くことができますが，マクロの展開はすべての定義の右辺値の評価に先立って行われることになっています．そのため，マクロの展開に必要な手続き等がマクロの展開が終わるまで使えない，といった問題が起こります．

[2] #`は quasisyntax の構文糖衣です．

そこで，R[6]RS では，Scheme コードの解釈を次のように規定しました．

- Scheme コードは，ライブラリ本体，あるいはトップレベルプログラム本体を 1 つの「ユニット」として解釈される．
- ユニットの解釈は，「expand フェーズ」と「run フェーズ」からなる．
- 各フェーズで使用するライブラリを指定できる．

そのため，マクロの展開内で使用される手続きは，同一のライブラリに定義されていることはできません．

また，マクロトランスフォーマについての定義の変更や，構文オブジェクトという概念の追加がなされました．構文オブジェクトは，構文の構造に加えて，その文脈情報を保持した Scheme のフォームを表現したものです．

第 III 章

SC, RSC, ER, IR について

1　Syntactic Closure

シンタクティッククロージャ

> ハイジ「Twitter を見てると，『信託ティッククロージャ』と呟く Schemer がちらちらいるわ」
> クララ「そうね．私も見かけたことがあるわ」
> ハイジ「ところで，それはなんなの？」
>
> アリスプの少女　ハイジニックマクロ　第 10 話

Lisper なら誰もが知っている概念として，クロージャ（関数閉包）という概念があります．Scheme においては，S 式と環境とをくっつけたものとして考えられます．ところで，これは，II 章の 2 節で述べた構文オブジェクトと似ています．シンタクティッククロージャも同様の考え方を用います．

通常，クロージャは lambda フォームが呼ばれたその場所での環境を持つことになりますが，シンタクティッククロージャは，任意の環境をその中に閉じ込めることができます．シンタクティッククロージャを作るには，make-syntactic-closure 手続きを使います．定義は次のとおりです．

```
(make-syntactic-closure <free-env> <free-vars> <expression>)
; => #<syntactic closure>
```

これを元に，sc-macro-transformer 手続きによって，マクロトランスフォーマを作ります．Chibi Scheme や Picrin では，次のように定義されています．

```
(define (sc-macro-transformer f)
  (lambda (expr use-env mac-env)
    (make-syntactic-closure mac-env '() (f expr use-env))))
```

これを用いた swap!マクロと定義は次のようになります.

```
(define-syntax swap!
  (sc-macro-transformer
   (lambda (form env)
     (let ((a (make-syntactic-closure env '() (second form)))
           (b (make-syntactic-closure env '() (third form))))
       `(let ((value ,a))
          (set! ,a ,b)
          (set! ,b value))))))
```

そして，sc-macro-transformer を適用すると，次のようになります.

```
(define-syntax swap!
  (lambda (form use-env mac-env)
    (let ((a (make-syntactic-closure use-env '() (second form)))
          (b (make-syntactic-closure use-env '() (third form))))
      (make-syntactic-closure mac-env '() `(let ((value ,a))
                                             (set! ,a ,b)
                                             (set! ,b value))))))
```

ここで，define-syntax は，マクロの名前となる識別子と，マクロトランスフォーマを受け取り，マクロ定義をします．マクロトランスフォーマは，第 1 引数にマクロ呼び出しの式を渡し，第 2・第 3 引数にそれぞれマクロ呼び出し時の環境と，マクロ定義時の環境とを渡します.

引数 a と b に渡された式のみが，マクロ呼び出し時の環境でのシンタクティッククロージャになっていることがわかります．(swap! x y) をマクロ展開した式を，環境がわかるように示してみると，次のようになります.

```
(#mac-env#let ((#mac-env#value #use-env#x))
  (#mac-env#set! #use-env#x #use-env#y)
  (#mac-env#set! #use-env#y #mac-env#value))
```

このようにして，健全性を確保することができます.

意図的な健全性の破壊

> ハイジ「Stallman の歌う，自由ソフトウェアの歌は癖になるわ」
> おんじ（最近の若いもんは妙な趣味をしとるな…）
> クララ「Join us now and share the software〜♪」
>
> アリスプの少女　ハイジニックマクロ　第 11 話

make-syntactic-closure 手続きの引数からもわかるとおり，syntax-case 同様 SC も健全性を意図的に破ることができます．例のごとく，aif の定義を見てみましょう．

```
(define-syntax aif
  (sc-macro-transformer
   (lambda (form env)
     (let ((pred (make-syntactic-closure env '() (second form)))
           (then (make-syntactic-closure env '(it) (third form)))
           (else (if (null? (cdddr form))
                     #f
                     (make-syntactic-closure env '() (fourth form)))))
       `(let ((it ,pred))
          (if it ,then ,else))))))
```

このようにすることで，aif の第 2 引数の中で it を使うことができるよう，シンボルを注入することができます．

identifier?と identifier=?

> ハイジ「あら，ユキちゃんじゃない？どうしたのかしら？」
> クララ「ハイジ，あなた Lisp エイリアンが見分けられるの？すごいわ！」
>
> アリスプの少女　ハイジニックマクロ　第 12 話

マクロを実装している際，cond における else のように，そのマクロ内での予約語のようなものが必要になる時があります．例えば，SRFI-26 の cut マクロでは，<>や<...>を予約語として持ちます．その cut マクロを SC で実装すると，次のようになります．

```
(define-syntax cut
  (sc-macro-transformer
   (lambda (form env)
     `(cut% () () ,@(map (lambda (ex)
                           (make-syntactic-closure env '() ex))
```

```
                                    (cdr form))))))

  (define-syntax cut%
    (sc-macro-transformer
     (lambda (form env)
       (let ((slots (cadr form))
             (combi (caddr form))
             (se (cdddr form)))
         (define (id=? x y)
           (and (identifier? x)
                (identifier=? env x env y)))
         (cond ((null? se)
                (let ((slots (reverse slots))
                      (combi (reverse combi)))
                  `(lambda ,slots ((begin ,(car combi)) ,@(cdr combi)))))
               ((id=? (car se) '<...>)
                (let ((slots (reverse slots))
                      (combi (reverse combi))
                      (rest-slot (make-syntactic-closure env '() 'rest-slot)))
                  `(lambda (,@slots ,@rest-slot) (apply ,@combi ,rest-slot))))
               ((id=? (car se) '<>)
                (let ((x (make-syntactic-closure env '() 'x)))
                  (let ((slots (cons x slots))
                        (combi (cons x combi)))
                    `(cut% ,slots ,combi ,@(cdr se)))))
               (else
                (let ((combi (cons (car se) combi)))
                  `(cut% ,slots ,combi ,@(cdr se)))))))))
```

SC を提供する処理系は，identifier?手続きと identifier=?手続きを同時に提供しており，これを用いてマクロ呼び出し内で予約語かどうかチェックします．identifier?と identifier=?の定義は次のとおりです．

```
(identifier? <object>)
; => #t or #f

(identifier=? <environment1> <identifier1> <environment2> <identifier2>)
; => #t or #f
```

cut マクロの実装を読むのがしんどい人のために，簡単な cond の実装も載せておきます[1].

```
(define-syntax cond-simple
  (sc-macro-transformer
   (lambda (form use-env)
     (capture-syntactic-environment
      (lambda (mac-env)
        (define (id? x y)
          (and (identifier? x)
               (identifier=? use-env x mac-env y)))
        (define (close-sc x)
          (make-syntactic-closure use-env '() x))
        (let ((clauses (cdr form)))
          (if (null? clauses)
              `(values)
              (let* ((fst (car clauses))
                     (rst (cdr clauses))
                     (test (car fst)))
                (if (id? test 'else)
                    `(begin ,@(map close-sc (cdr fst)))
                    `(if ,(close-sc test)
                         (begin ,@(map close-sc (cdr fst)))
                         (cond-simple ,@rst)))))))))))
```

2 Reversed Syntactic Closure

裏返し

> ハイジ「ペーターってば，最近いつもクララのこと見てるわよね」
> ペーター「べ，別にそんなことねえよ／／／／」
> クララ「その態度は愛情の裏返しだと思うけど，男のツンデレはキモいだけよ？」
>
> アリスプの少女　ハイジニックマクロ　第 13 話

Syntactic Closure の理解が済んでいれば，Reversed Syntactic Closure の理解は簡単です．

[1] capture-syntactic-environment 手続きは，MIT/GNU Scheme や CHICKEN Scheme に用意されています．

RSC では rsc-macro-transformer でマクロトランスフォーマを作りますが，1 節同様，その実装を見てみましょう．

```
(define (rsc-macro-transformer f)
  (lambda (expr use-env mac-env)
    (make-syntactic-closure use-env '() (f expr mac-env))))
```

sc-macro-transformer とは対照的に，引数として渡される手続きにマクロ定義時の環境を渡していることがわかります．そのため，swap!の定義は次のようになります．

```
(define-syntax swap!
  (rsc-macro-transformer
   (lambda (form env)
     (let ((a (second form))
           (b (third form))
           (let-r (make-syntactic-closure env '() 'let))
           (value (make-syntactic-closure env '() 'value))
           (set!-r (make-syntactic-closure env '() 'set!)))
       `(,let-r ((,value ,a))
          (,set!-r ,a ,b)
          (,set!-r ,b ,value))))))
```

3 Explicit Renaming

手続きという API

> クララ「この時期は鼻がムズムズするわね」
> ハイジ「クララったら，花粉症なのね？」
> クララ「そうなのよ．ふぁっ…ふぁっ…ふぁんくしょん！！」
>
> アリスプの少女　ハイジニックマクロ　第 14 話

Explicit Renaming は，その名のとおり，明示的に Rename するマクロです．SC や RSC と同様，er-macro-transformer でマクロトランスフォーマを作ります．er-macro-transformer は，マクロフォームと，シンボルを受け取って，マクロ定義時の環境でそのシンボルが示すものと同じものを指す別の識別子を返す手続き，そして 2 つの識別子を引数に取り，マクロ展開時の環境で同じものを示す識別子であるか比較する手続きの 3 つを，その引数である手続きに渡します．日本語が入り組んで非常にわかりにくいので，コードで示したいと思います．例のごとく，swap!です．

```
(define-syntax swap!
  (er-macro-transformer
   (lambda (form rename compare?)
     (let ((a (second form))
           (b (third form)))
       (,(rename 'let) ((,(rename 'value) ,a))
         (,(rename 'set!) ,a ,b)
         (,(rename 'set!) ,b ,(rename 'value)))))))
```

見た目は，RSC 版の swap! と似ています．ER は SC や RSC から 1 段階抽象化して，環境そのものを渡すのではなく，Rename する手続きとして渡しています．ちなみに，その Rename するための手続きは，純関数なので，同じシンボルを与える限り，同じ識別子を返します．

しないをする選択

> ハイジ「しないでするスポーツってな〜んだ？」
> ペーター「なんだろうな…？」
> クララ「なにかしら…？わからないわ．ハイジ，答えを教えてちょうだい」
> ハイジ「竹刀でするから，剣道よ！」
> おんじ（ここはヨー○ッパじゃぞハイジ…）
>
> アリスプの少女　ハイジニックマクロ　第 15 話

ER でも健全性を意図的に破ることができ，それは単に Rename しないことで可能です．ER での aif の実装を見てみましょう．

```
(define-syntax aif
  (er-macro-transformer
   (lambda (form rename compare?)
     (let ((pred (second form))
           (then (third form))
           (else (if (null? (cdddr form)) #f (fourth form))))
       `(,(rename 'let) ((it ,pred))
          (,(rename 'if) it ,then ,else))))))
```

Rename せずに it を挿入することで，簡単に aif が定義できます．

コンパリスン

> ハイジ「good，better，best…」
> クララ「英語のお勉強ね？頑張ってるわね」
> ハイジ「bad，worse，worst…」
> おんじ「よし，よろし，わろし，あし…」
>
> アリスプの少女　ハイジニックマクロ　第 16 話

ER 実装の cond-simple と cut を載せておきます．ちなみに，CHICKEN のドキュメントでは，identifier?の代わりに symbol?を用いています．

```
(define-syntax cond-simple
  (er-macro-transformer
   (lambda (form rename compare?)
     (define (cmp? x y)
       (and (identifier? x)
            (compare? x y)))
     (let ((clauses (cdr form)))
       (if (null? clauses)
           `(,(rename 'values))
           (let* ((fst (car clauses))
                  (rst (cdr clauses))
                  (test (car fst)))
             (if (cmp? test 'else)
                 `(,(rename 'begin) ,@(cdr fst))
                 `(,(rename 'if) ,test
                   (,(rename 'begin) ,@(cdr fst))
                   (,(rename 'cond-simple) ,@rst)))))))))
```

cut は以下．

```
(define-syntax cut-er
  (er-macro-transformer
   (lambda (form rename compare?)
     `(,(rename 'cut%-er) () () ,@(cdr form)))))

(define-syntax cut%-er
  (er-macro-transformer
```

```
(lambda (form rename compare?)
  (define (cmp? x y)
    (and (identifier? x)
         (compare? x y)))
  (let ((slots (cadr form))
        (combi (caddr form))
        (se (cdddr form)))
    (cond ((null? se)
           (let ((slots (reverse slots))
                 (combi (reverse combi)))
             `(,(rename 'lambda) ,slots
               ((,(rename 'begin) ,(car combi)) ,@(cdr combi)))))
          ((cmp? (car se) (rename '<...>))
           (let ((slots (reverse slots))
                 (combi (reverse combi))
                 (rest-slot (rename 'rest-slot)))
             `(,(rename 'lambda) (,@slots ,@rest-slot)
               (,(rename 'apply) ,@combi ,rest-slot))))
          ((cmp? (car se) (rename '<>))
           `(,(rename 'cut%-er)
             ,(cons (rename 'x) slots)
             ,(cons (rename 'x) combi)
             ,@(cdr se)))
          (else `(,(rename 'cut%-er)
                  ,slots
                  ,(cons (car se) combi)
                  ,@(cdr se)))))))
```

4 Implicit Renaming

よしなに

クララ「いちいち gensym するの面倒だわ」
おんじ「いいものがあるぞ」
クララ「with-gensym かしら？」
おんじ「もっといいものだ」

アリスプの少女　ハイジニックマクロ　第 17 話

ER に対応する形で，Implicit Renaming があります．名前のとおり，IR は暗黙的に Rename します．そのため，ir-macro-transformer 手続きは，Rename させないための手続きを与えます．IR での swap!の実装は次のとおりです．

```
(define-syntax swap!
  (ir-macro-transformer
   (lambda (form inject compare?)
     (let ((a (second form))
           (b (third form)))
       `(let ((value ,a))
          (set! ,a ,b)
          (set! value ,b))))))
```

意図的に健全性を破る際には，ir-macro-transformer から渡される，第 2 引数の手続きを使うだけです．以下は IR での aif です．

```
(define-syntax aif
  (ir-macro-transformer
   (lambda (form inject compare?)
     (let ((pred (second form))
           (then (third form))
           (else (if (null? (cdddr form)) #f (fourth form)))))
       `(let ((,(inject 'it) ,pred))
          (if ,(inject 'it) ,then ,else))))))
```

IR は非常に便利ですが，すべての識別子を自動的に Rename するため，マクロ展開の計算オーダーが高いという欠点があります．

あとがき

前回，Lisp のトラディショナルなマクロについて書かせていただきました．今回は，Scheme のハイジニックマクロについてでしたが，実は，前回の部誌を公開した，文化祭の打ち上げの時点でこの内容について書くことを決めていました．そのため，色々なところに回って宣伝することが出来ました．ですが，実際執筆し始めると，自分はこんなに文章がかけないものかと辟易するほどでした．結局，締め切りを 2 度も引き伸ばした上，更に 3 日オーバーするという惨憺たる結果でした．

今回のタイトル，「アリスプの少女ハイジニックマクロ」というのは，今回のテーマが決まって数日後にあった修学旅行の行きの飛行機で思いついたものです．その時は非常に自信があったのですが，1 年近くたった今，自分の中ではすっかり風化してしまい，全く面白くないものになってしまいました．

Scheme は，他の言語に比べて日本語のドキュメントが非常に少ないと思います．この記事を書くことで，Scheme の日本語ドキュメントを増やし，ひいては日本の Schemer が増やすことに貢献できればと思います．

参考文献

[1] anonymous. syntactic closure. http://community.schemewiki.org/?syntactic-closures, September 2010.

[2] The Chicken Team, http://wiki.call-cc.org/man/4/The User's Manual. *The CHICKEN User's Manual*.

[3] William Clinger. Hygienic macros through explicit renaming. *SIGPLAN Lisp Pointers*, Vol. IV, No. 4, pp. 25–28, October 1991.

[4] Chris Hanson. *MIT/GNU Scheme Reference Manual*. http://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref.pdf, October 2010.

[5] Shiro Kawai. Scheme のマクロ. http://blog.practical-scheme.net/shiro?20100425-scheme-macro, April 2010.

[6] kei. Yet another syntax-case explanation. http://compassoftime.blogspot.jp/2013/05/yet-another-syntax-case-explanation.html, May 2013.

[7] Richard Kelsey, William Clinger, and Jonathan (eds.) Rees. The revised^5 report on the algorithmic language scheme. Technical report, Scheme Steering Committee, http://www.schemers.org/Documents/Standards/R5RS/, August 1998.

[8] Oleg Kiselyov. Low- and high-level macro programming in scheme. http://okmij.org/ftp/Scheme/macros.html, July 2013.

[9] leque. 様々な hygienic macro. http://d.hatena.ne.jp/leque/20080528/p1, May 2008.

[10] Sari Sakaue and Kenichi Asai. 対称 λ 計算の基礎理論. 第 10 回プログラミングおよびプログラミング言語ワークショップ (PPL2008), March 2008.

[11] Alex Shinn. [chicken-users] macro systems and chicken (long). http://lists.gnu.org/archive/html/chicken-users/2008-04/msg00013.html, April 2008.

[12] Alex Shinn, John Cowan, and Arthur A. Gleckler (eds.). The revised^7 report on the algorithmic language scheme. Technical report, Scheme Steering Committee, http://scheme-reports.org/, July 2013.

[13] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton Van Straaten (eds.). The revised^6 report on the algorithmic language scheme. Technical report, Scheme Steering Committee, http://www.r6rs.org/, September 2007.

[14] Philip Wadler. Call-by-value is dual to call-by-name. *SIGPLAN Not.*, Vol. 38, No. 9, pp. 189–201, August 2003.

[15] wasabiz. syntactic closure は内部でどのように実装されているのか. http://wasabiz.hatenablog.com/entry/20120331/1333209310, March 2012.