プログラミング言語論第13回
マクロ

情報工学科 山本修身

マクロ：プログラムの書き換え規則 (1)

マクロとは，プログラムを書き換えるための規則のことである．単純なマクロの例はすでにCに見ることができる．Cにはcppというマクロプロセッサが標準で用意されている．cppへの命令はすべて#から始まる．cppが解釈する文法はCの文法とは関係ない．一般にCのプログラムには2種類の文法が混在している．

```
#define dd(x) x x
#define dd2(x) dd(dd(x))
#define dd3(x) dd2(dd2(x))
#define dd4(x) dd3(dd3(x))

dd4(a)
```

↓ cpp

```
a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a a
```

マクロ：プログラムの書き換え規則 (2)

Cのプリプロセッサはいくつかの命令からなる．すべて#からはじまる．書き換えを定義する命令は#defineであり，変数によって変形規則を記述することができる．

```
#define foo(x) x + 2

foo(y) / 2
```

というプログラムは，以下のように書き換えられる．

```
y + 2 / 2
```

書き換えられた結果を良く考えないと想定外の結果を導いてしまうことがある．Cの場合，プリプロセッサがコメント文の処理やヘッダファイルの読み込み，条件コンパイルなどの機能を同時に担う．

マクロは非常に強力であるが，同時にバグの原因となるので，不必要な状況で利用するのは避けるべきである．C++では#defineはほとんど必要なくなっている．

C/C++のマクロの注意点

マクロ全般に言えるが，マクロは文字列を文字列に変える行為なので，文字列として思わぬ書き換えが起こる可能性があり，それを意識する必要がある．括弧などをつかって想定外のことが起こらないように注意する．

```
#include <stdio.h>

#define sqr(x) x * x
#define sqr2(x) ((x) * (x))

int main(){
  printf("%d-%d\n", sqr(3 + 4), sqr2(3 + 4));
  return 0;
}
```

↓

```
OMacBook:yama605> !gcc
gcc macro3.c -o macro3
OMacBook:yama606> ./macro3
19-49
```

Schemeにおける古典的マクロ (1)

Schemeにおけるマクロは多少見え方が異なる．<u>**マクロは与えられた引数を評価せずに渡し，ボディーを2回続けて現在評価している環境で評価する．**</u>これがマクロの動作のすべてである．この動作が理解出来ていれば，古典的なマクロについては完全に理解出来ていると言える．

```
(define-macro (foo x) (list '* x x))
(display (foo 3))(newline)
```

↓

```
OMacBook:yama502> kawa test-macro1.scm
9
```

マクロはdefineの関数定義と同様に定義する．しかし，評価の方法が異なる．`(list '* 3 3)`を一度評価すると，`(* 3 3)`になる．これをもう一度評価すると9となる．したがって，結果を9が表示される．この呼び出し方の場合，関数と変わらない．

Schemeにおける古典的マクロ (2)

Schemeでは`lambda`が使えるので，多少見栄えが悪くても多くのことは`lambda`を用いて表現することができる．その例としては前半の講義で用いた`myfor`などがある．すなわち，`lambda`で包むことによって評価を遅らせて関数に中身を渡すことができる．

Schemeにおけるマクロは単なる評価手順の違いで説明される．引数として渡されるとき，評価されないことが重要である．

以前のスライドで定義した`myfor`は以下のようなものであった．それを右下のようにかけるようにしたい．

```
(define (myfor i n func)
  (if (= i n) '()
      (begin
    (func i)
          (myfor (+ i 1) n func))))
```

```
(myfor 0 10 (lambda (n)
              (display n)
              (newline)))
```

→

```
(for n 0 10
      (display n)
      (newline)))
```

Schemeにおける古典的マクロ (3)


```
(myfor 0 10 (lambda (n)
      (display n)
      (newline)))
```

⟷

```
(for n 0 10
     (display n)
     (newline))
```

上記のようにforを書き換えるように以下のようにdefine-macroでマクロの定義をする.

```
(define (myfor i n func)
  (if (= i n) '()
      (begin
        (func i)
        (myfor (+ i 1) n func))))

(define-macro (for n from to . body)
  (list 'myfor from to
        (cons 'lambda (cons (list n) body))))
```

マクロでは引数 n, from, to, body は評価されず，そのまま本体に展開される．もちろんbodyはtoよりも後ろにくるデータを要素 t するリストになる.

Schemeにおける古典的マクロ (4)

ここで定義したforを用いれば，九九の表を出力するプログラムは以下のようになる.

```
(define (kuku n)
  (for i 0 n
       (for j 0 n
            (let ((k (* i j)))
              (display (if (< k 10) "  " " "))
              (display k)))
       (newline)))

(kuku 10)
```

→

```
 0  0  0  0  0  0  0  0  0  0
 0  1  2  3  4  5  6  7  8  9
 0  2  4  6  8 10 12 14 16 18
 0  3  6  9 12 15 18 21 24 27
 0  4  8 12 16 20 24 28 32 36
 0  5 10 15 20 25 30 35 40 45
 0  6 12 18 24 30 36 42 48 54
 0  7 14 21 28 35 42 49 56 63
 0  8 16 24 32 40 48 56 64 72
 0  9 18 27 36 45 54 63 72 81
```

マクロのためのクオート

マクロを定義する場合，式をlistやconsなどで作ると間違いが起こりやすい．そこで，以下の3つの特殊形式を導入することによって，直感的なプログラミングを可能にする：

quasiquote, unquote, unquote-splicing

これらの特殊形式はそれぞれ「` （バッククオート）」 「,」 「,@」で表現することできる．前述のforは以下のようなマクロ定義として書くことができる.

```
(define-macro (for n from to . body)
  `(myfor ,from ,to (lambda (,n) ,@body)))
```

バッククオート ` は基本的にクオートと同じであるが，内部あるunquote (,) やunquote-splicing (,@) を展開する．,@ はリストを展開するとき，周りの()を取り去ってから式に埋め込む.

マクロの例 (1)

Schemeにwhile式を定義してみる．これは，Cのwhileのように条件が成り立つ間，本体を繰り返し実行させるものである．まず，どのように書き換えるかを決める.

(while 条件式 本体) という形の式を考え，本体は式が複数並ぶことを許すことにする．この式を以下のように書き換える．ここでletで外側を囲っているのは，新たなフレームを作って，そこにloop-funcの束縛を作るためである（whileが使われる環境の直接のフレームで別にloop-funcが定義されているとまずいため）.

```
(let ()
  (define (loop-func)
    (if 条件式
        (begin 本体 loop-func)))
  (loop-func))
```

マクロの例 (2)

以下のようにwhileマクロを定義することによってwhileが使えるようになる.

```
(define-macro (while cond . body)
  `(let ()
     (define (loop-func)
       (if ,cond
           (begin ,@body (loop-func))
           '()))
     (loop-func)))
```

```
(let ((i 0))
  (while (< i 10)
         (display i)
         (newline)
         (set! i (+ i 1))))
```

→

```
0
1
2
3
4
5
6
7
8
9
```

マクロが評価される環境について

以前説明したように関数の評価では，その関数が定義された環境に引数用のフレームを付加した環境で本体を評価した．マクロの場合には，マクロが定義された環境ではなくて，現在実行されている環境で評価される.

マクロ定義

```
(define m 30)
(define-macro (dispm)
  `(begin (display m)(newline)))
(let ((m 20))
  (dispm))
(dispm)
```

→

```
20
30
```

マクロは動的スコーピングルールを用いているとも言える

関数定義

```
(define m 30)
(define (dispm)
  (display m)
  (newline))
(dispm)
(let ((m 20))
  (dispm))
```

→

```
30
30
```

関数は常に静的スコーピングルールを用いる.

評価戦略について (1)

Schemeが式を評価する際，関数に引数を適用する方法は基本的にまず引数のそれぞれを評価して値にしてから，それを関数に渡す方法である．このような関数適用の方法は**正格評価 (strict evaluation)** と呼ばれる．これは値呼びまたは**最内簡約 (innermost reduction)** とも呼ばれる．正格評価では，式の評価は以下のように進む

```
(define (add x y) (+ x y))

(define (mult x y) (* x y))

(add (mult 2 3) (mult 1 2)) -> (add (* 2 3) (mult 1 2)) -> (add 6 (mult 1 2)) -> (add 6 (* 1 2)) (add 6 2) -> (+ 6 2) -> 8
```

Scheme で書かれたSchemeインタプリタのソースコードを眺めるとこの順に評価が行われているのがわかる．

評価戦略について (2)

これに対して，ここで示したマクロなどは渡された引数を評価せずにそのまま本体に渡して後で評価を実行する．このような評価方法を**非正格評価 (non-strict evaluation)**と呼ぶ．もちろん式は最終的には評価される．特に一番外側から評価していく方法を**最外簡約 (outermost reduction)** と呼ぶ．

```
(add (mult 2 3) (mult 1 2)) -> (+ (mult 2 3) (mult 1 2)) -> (+ (* 2 3) (mult 1 2)) -> (+ (* 2 3) (* 1 2)) -> (+ 6 (* 1 2)) -> (+ 6 2) -> 8
```

この関数適用の方法を**名前呼び (call by name)** と呼ぶ．副作用がない式のみを用いた場合には正格評価でも非正格評価でも結果は変わらない．

引数を評価しないSchemeのマクロも非正格評価をしていると言える．

評価戦略について (3)

最外簡約の場合，引数は全く評価されず必要に応じて評価される．以下のような例では最内簡約と実行の様子が異なる．

```
(define (add x y) (+ x y))

(define (sqr x) (* x x))
```

call by value

```
(sqr (add 2 3)) -> (sqr (+ 2 3)) -> (sqr 5) -> (* 5 5) -> 25
```

call by name

```
(sqr (add 2 3)) -> (* (add 2 3) (add 2 3)) -> (* (+ 2 3) (add 2 3)) -> (* 5 (add 2 3)) -> (* 5 (+ 2 3)) -> (* 5 5) -> 25
```

call by nameでは，引数が関数本体で使われなければ，結局その引数は1回も評価されない．

現代的なマクロ定義： define-syntax (1)

現在のSchemeではより一般的なマクロを利用することができる．より複雑なマクロが効率よく定義できる．

まず，ある変数に0をセットするマクロを考える．

```
(define-macro (set-zero! x)
  `(set! ,x 0))

(let ((m 10))
  (set-zero! m)
  (display m)
  (newline))
```

これを現代的なマクロで定義すると以下のようになる．

```
(define-syntax set-zero!
  (syntax-rules ()
    ((_ x)
     (set! x 0))))

(let ((m 10))
  (set-zero! m)
  (display m)
  (newline))
```

現代的なマクロ定義： define-syntax (2)

define-syntaxでマクロを定義する場合，つぎのように書く．

```
(define-syntax マクロ名
   (syntax-rules 予約語リスト
      (パターン1 テンプレート1)
      (パターン2 テンプレート2)
       ........
      (パターンn テンプレートn)))
```

予約語を使わない場合には予約語リストは()となる．パターンはマクロのパターンで (_ x1 x2) のようなものである．_は任意のデータ．前述の例の場合，パターンは (_ x) であり，これは (set-zero! m)の形に対応する．set-zero!が_に，mがxにマッチする．マッチした結果をテンプレートに埋め込んだものが評価される．前述の例の場合，(set! x 0)となる．上から順にマッチするものが使われる．

多少見やすいforを作る

今までのforよりも多少みやすくて，少しだけ高機能なforを定義してみる．

```
(define-syntax for
 (syntax-rules (from to downto)
   ((_ i from s to t body ...)
    (let loop ((i s))
      (if (= i t) '()
          (begin
            body ...
            (loop (+ i 1))))))
   ((_ i from s downto t body ...)
    (let loop ((i s))
      (if (= i t) '()
          (begin
            body ...
            (loop (- i 1))))))))
```

```
(define (kuku n)
  (for i from 1 to (+ n 1)
       (for j from 1 to (+ n 1)
            (let ((k (* i j)))
              (display (if (< k 10) "   " "  "))
              (display k)))
       (newline)))
(kuku 9)
```

```
 1  2  3  4  5  6  7  8  9
 2  4  6  8 10 12 14 16 18
 3  6  9 12 15 18 21 24 27
 4  8 12 16 20 24 28 32 36
 5 10 15 20 25 30 35 40 45
 6 12 18 24 30 36 42 48 54
 7 14 21 28 35 42 49 56 63
 8 16 24 32 40 48 56 64 72
 9 18 27 36 45 54 63 72 81
```

```
(for i from 0 to 10
     (display i))
(newline)

(for i from 10 downto 0
     (display i))
(newline)
```

→

```
0 1 2 3 4 5 6 7 8 9
10 9 8 7 6 5 4 3 2 1
```

...は後ろのその他すべての要素を表す．

再帰するマクロ

マクロを再帰させて順番に中身を評価していくmy-andを定義してみる．この場合，my-andのマクロ定義の中でmy-andが使われている．

```
(define-syntax my-and
  (syntax-rules ()
    ((_) #t)
    ((_ cond) cond)
    ((_ cond1 cond2 ...)
     (if cond1 (my-and cond2 ...) #f))))
```

```
(let ((x 2) (y 4) (z 5))
  (display
   (if (my-and (begin (display 'here1) (= (* x x) y))
               (begin (display 'here2) (= x y))
               (begin (display 'here3) (= (- z 1) y))) 'ok 'ng))
  (newline))
```

```
here1 here2 ng
```