# プログラミング入門
# 「メタサーキュラーインタプリタ」

二宮 崇（ ninomiya@cs.ehime-u.ac.jp ）

教科書

Structure and Interpretation of Computer Programs, 2nd Edition: Harold Abelson, Gerald Jay Sussman, Julie Sussman, The MIT Press, 1996

## 超言語的抽象（第 4 章）

### 4.1 超循環評価器

4 章では、超循環評価器（メタサーキュラーインタプリタ, metacircular interpreter）について学びます。メタサーキュラーインタプリタとは、同じ言語で記述されたインタプリタのことで、ここでは Scheme を使うので、Scheme で Scheme インタプリタを作る、というわけです。Scheme で Scheme インタプリタを作ることにどういう意味があるか、ということはわかりにくいですが、自分のための Scheme インタプリタを作れる、ということです。例えば、自分で Scheme の方言を作ることができるし、Scheme を拡張して、例えば、型付 Scheme であったり、オブジェクト指向 Scheme を作ったり、いろいろな拡張ができます。他にも、正規順序で評価する Scheme インタプリタ、遅延評価で必ず評価するインタプリタなど、様々な動きの Scheme インタプリタを作ることができます。構文解析器と組み合わせれば、他のプログラミング言語のインタプリタも実装することができるようになります。また、S 式を表現して処理するライブラリが例えば C 言語にあれば、ここで紹介するプログラムを C 言語に移植して、C 言語で Scheme インタプリタを実装することができるわけで、そういう意味ではインタプリタを作成する基本技術を学ぶということにもなります。

まず、完成されたインタプリタの動きをみて、どういう風に動作するのか理解しましょう。

```
> (define genv (setup-environment))
> genv
(((false true car cdr cons null? + - * / remainder < > <= >=)
  #f
  #t
  (primitive #<procedure:mcar>)
  (primitive #<procedure:mcdr>)
  (primitive #<procedure:mcons>)
  (primitive #<procedure:null?>)
  (primitive #<procedure:+>)
  (primitive #<procedure:->)
  (primitive #<procedure:*>)
  (primitive #<procedure:/>)
  (primitive #<procedure:remainder>)
  (primitive #<procedure:<>)
  (primitive #<procedure:>>)
  (primitive #<procedure:<=>)
  (primitive #<procedure:>=>)))
```

最初に setup-environment と実行していますが、これは、環境を作成する関数で、これから作るインタプリタ内の変数とその値のペアを格納しています。インタプリタが使う最も原始的な関数と true と false だけが定義されています。ここでは、環境は二つのリストのペアになっていて、それぞれ、変数のリストと変数に対応する値のリストになっています。

続いて次のように実行できます。インタプリタに入力するプログラムはクォートによるシンボルのリストになっていることに注意してください。

```
> (myeval '(define a 5) genv)
ok
```

ここで、genv という環境で a に 5 という値を対応づけました。すると、

```
> (myeval '(+ a 10) genv)
15
```

という感じで、a に格納された値(=5)を使って(+ 5 10)を評価し、15 という値を返しています。他にも、

```
> (myeval '(if (< a 10) 20 5) genv)
```

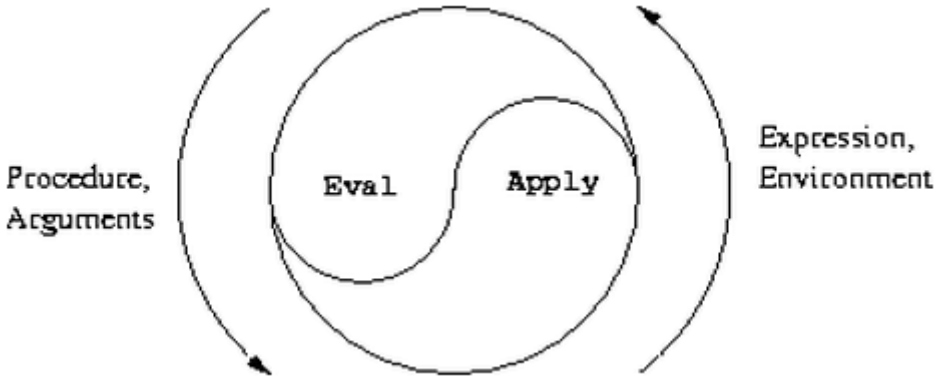
```
20
```

という感じで、if 文を評価できていることがわかります。さらに、

```
> (myeval '(define (f x) (lambda (y) (+ x y))) genv)
ok
> (myeval '(define g (f 100)) genv)
ok
> (myeval '(g 200) genv)
300
```

とこのように、ラムダ式を返すことができていることがわかります。

インタプリタは二つの基本処理から実現されています。一つは eval で、もう一つは apply です。eval は式の解釈を行いインタプリタ全体の動作を制御します。apply は与えられた演算子と引数から、その引数を演算子に適用するということを行います。eval と apply を交互に実行することによって、インタプリタは動作します。




### 4.1.1 評価器の中核

まず、呼び出しの元になっている eval をみてみましょう。ただし、元々Scheme に定義されている eval とぶつかるとだめなので、ここでは myeval という名前にしています。

#### Eval

```
(define (variable? exp) (symbol? exp))
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
```

```
      #f))
(define (application? exp) (pair? exp))


(define (myeval exp env)
  (cond ((number? exp) exp)
        ((string? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((tagged-list? exp 'define) (eval-definition exp env))
        ((tagged-list? exp 'if) (eval-if exp env))
        ((tagged-list? exp 'lambda) (eval-lambda exp env))
        ((application? exp) (eval-application exp env))
        (else (display "Unknown expression type -- EVAL") (display exp) (newline))))
```

ここで myeval に注目すると、exp という式と env という環境を受け取ります。exp という式をみたとき、この式が数や文字列なら、その値をそのまま返します。次に変数だったら、その変数にバインドされている値を lookup-variable-value で環境の中から探し出します。つづいて、特殊形式で、define, if, lambda だったらそれぞれ eval-definition, eval-if, eval-lambda を実行しなさい、とだけ書いています。最後に appication?とありますが、これは、この式がペアになっている、つまり、リストになっている場合の処理になります。これはつまり演算子や引数を含む一般の式の処理になります。これに対しては、eval-application が実行されます。eval の全体像はこんな感じになっています。

続いて、各 eval-definition, eval-if, eval-lambda が何をしているかみてみましょう。

## Eval-definition

```
(define (eval-definition exp env)
  (define-variable!
    (definition-variable exp)
    (myeval (definition-value exp) env)
    env)
```

```
  'ok)

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp) ; variable  e.g. (define x 2)
      (caadr exp))) ; procedure   e.g. (define (f x) (+ x 1))

(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp) ; variable
      (cons 'lambda (cons (cdr (car (cdr exp)))
                          (cdr (cdr exp))))))) ; procedure
```

これは、define-variable!というさらに基本的な関数で define が実装されている、ということがわかります。definition-variable は define で定義しようとしている変数を取り出します。definition-value は定義しようとしている変数の値を取り出します。ここで、define が(define <変数> <値>)という形をしているか、それとも(define (<関数名> <引数>…) <関数本体>)という形をしているか判断して、後者の場合は関数を lambda 式に変えて、変数にバインドする、ということを行っています。define-variable!が重要な関数になるわけですが、それについては環境の説明のところで紹介します。

次に、eval-if です。

## Eval-if

```
(define (if-pred exp) (cadr exp))
(define (if-then exp) (caddr exp))
(define (if-else exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      #f))
(define (eval-if exp env)
  (if (myeval (if-pred exp) env)
      (myeval (if-then exp) env)
```

```
    (myeval (if-else exp) env)))
```

これは、exp が if 文であったときは、述語部分をまず myeval で評価し、それが真であったならば、帰結部(if-then exp)を myeval で評価し返値とします。そうでなければ代替部(if-else)を myeval で評価し返値とします。

次は、eval-lambda です。

## Eval-lambda　e.g., (lambda (x y) (+ x y))

```
(define (eval-lambda exp env)
  (list 'procedure
        (car (cdr exp)) ; lambda-parameters
        (cdr (cdr exp)) ; lambda-body
        env))
```

これは、つまり、lambda 式にであったときは、それを procedure というタグがついた環境付のリストに変換するということを行っています。

最後に一般の式の評価についてです。

## Eval-application

```
(define (eval-application exp env)
  (let ((evaled-exp (map (lambda (x) (myeval x env)) exp)))
    (myapply (car evaled-exp) (cdr evaled-exp))))
```

一般の式の場合は、上記のように部分式を全て myeval で評価するということを行います。つまり、(+ (+ 3 2) (+ 4 5))とあったら、(+ 3 2)と(+ 4 5)に対し、myeval をまた相互再帰呼び出しで実行します。そのあと、myapply に対し、その評価ずみの演算子

と評価済みの引数を適用します。つまり、大部分の処理は myapply に引き継がれている、ということがわかります。

myeval の次に myapply を定義します。

## Apply

```
(define (eval-seq exps env)
  (cond ((null? (cdr exps)) (myeval (car exps) env))
        (else (myeval (car exps) env)
              (eval-seq (cdr exps) env))))

(define (myapply procedure arguments)
  (cond ((tagged-list? procedure 'primitive) (apply (cadr procedure) arguments))
        ((tagged-list? procedure 'procedure)
         (let ((procedure-parameters (car (cdr procedure)))
               (procedure-body (car (cdr (cdr procedure))))
               (procedure-environment (car (cdr (cdr (cdr procedure))))))
           (eval-seq
            procedure-body
            (extend-environment procedure-parameters arguments procedure-environment))))
        (else (display "Unknown procedure type -- APPLY") (display procedure) (newline))))
```

primitive とあるのは、+や-など組み込みで用意されている関数の場合です。myapply がうけとる procedure というのは、タグがつけられたリストになっていることを想定していて、primitive というタグがつけられていたら、その演算子の本体部分には組み込みの関数が入っていることを想定していて、それに対し、apply を実行します。ここでの apply は組み込みの特殊な関数で、以前にもでてきたことがありますが、リスト形式の引数を受け取って、(apply <演算子> (<引数 1> <引数 2> … <引数 n>))と実行すると、(<演算子> <引数 1> <引数 2> … <引数 n>)を実行したのと同じ結果が得られます。この apply を使いたくなければ primitive な演算子は必ず 1 項か 2 項しかとらない、というように制限をかける、という手もあります。

続いて、組み込みではない一般の場合(procedure)ですが、まず、この procedure の本体部分(procedure-body)が式の列(リスト)になっている場合を想定して、各要素を先頭から順に処理をしていって、最後の要素を処理したときにその結果を全体の結果として返す(eval-seq)ということを行います。ただし、このとき、新しいフレームを作って環境を拡張してから procedure を実行しています。新しいフレームでは、その関数の被演算子に引数を値として対応させています。

まとめると、apply は、組み込み述語に対応する関数なら、その組み込み述語に引数を適用して直接値を求めます。一般の述語の場合は、新しくフレームを作って環境を拡張して、その関数の本体を順に myeval で評価していく、ということを行います。

### 4.1.3 評価器のデータ構造

最後に、変数の値を格納する環境をどうやって実現するか、ということについて説明します。まず、簡単のためにフレームを作ったりフレームの中の変数や値のリストを得たり、追加したりする関数を定義します。

**フレーム**

```
(define (make-frame variables values) (cons variables values))
(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))
(define (add-binding frame var val)
  (set-car! frame (cons var (frame-variables frame)))
  (set-cdr! frame (cons val (frame-values frame))))
```

次に、新しいフレームを環境に追加する関数 extend-environment を定義します。

**環境にフレームを追加**

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (begin (display "Too many arguments! ") (display vars) (display " ") (display vals)
(newline))
```

```
          (begin (display "Too few arguments! ") (display vars) (display " ") (display vals)
(newline)))))
```

これは基本的には make-frame でつくる新しいフレームをリストである環境の先頭に追加する、ということを単純に行っています。ただし、変数の数とリストの数が一致していない場合は、本来その関数がとるべき被演算子の数と引数の数が一致していない、ということなので、エラーを返すべき、ということになります。

次に、束縛されている変数の値を得たい時のための lookup-variable-value を定義します。

## Lookup-variable-value

```
(define (lookup-variable-value var env)
  (define (scan vars vals)
    (cond ((null? vars) '())
          ((eq? (car vars) var) (list (car vals)))
          (else (scan (cdr vars) (cdr vals)))))
  (define (env-loop env)
    (if (eq? env '())
        (begin (display "Unbound variable") (display var) (newline))
        (let ((v (scan (frame-variables (car env))
                       (frame-values (car env)))))
          (if (null? v)
              (env-loop (cdr env))
              (car v)))))
  (env-loop env))
```

環境の先頭フレームから順にみていって、フレームの中に対応する変数があればその値を返します。次は、環境の中で変数を定義する場合です。

## Define-variable!

```
(define (define-variable! var val env)
```

```
  (let ((frame (car env)))
    (define (scan vars vals)
      (cond ((null? vars) (add-binding frame var val))
            ((eq? var (car vars)) (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame) (frame-values frame))))
```

これは、環境の中で一番先頭にきているフレームだけをみて、そこに対応する変数があれば、その変数の値を書き換えて、なければ、定義しようとしている変数とその値をフレームに追加(add-binding)します。

### 4.1.4 評価器をプログラムとして走らせる

最後に、初期状態の環境を作る関数 setup-environment を定義します。

**初期状態の環境**

```
(define (setup-environment)
  (let ((initial-env
         (cons (make-frame (list 'car 'cdr 'cons 'null? '+ '- '* '/ 'remainder '< '> '<= '>=)
                           (map (lambda (x) (list 'primitive x))
                                (list car cdr cons null? + - * / remainder < > <= >=)))
               '())))
    (define-variable! 'true #t initial-env)
    (define-variable! 'false #f initial-env)
    initial-env))
```

これで作られるデータ構造は、最初に説明したように二つのリストのペアになっていて、それぞれ変数のリスト、各変数に対応する値のリストとなっています。

# 付録

最後に metacircular interpreter のプログラムをまとめて掲載しておきます。

```
(define (variable? exp) (symbol? exp))
(define (tagged-list? exp tag)
  (if (pair? exp)
      (eq? (car exp) tag)
      #f))
(define (application? exp) (pair? exp))

;;; eval
(define (myeval exp env)
  (cond ((number? exp) exp)
        ((string? exp) exp)
        ((variable? exp) (lookup-variable-value exp env))
        ((tagged-list? exp 'define) (eval-definition exp env))
        ((tagged-list? exp 'if) (eval-if exp env))
        ((tagged-list? exp 'lambda) (eval-lambda exp env))
        ((application? exp) (eval-application exp env))
        (else (display "Unknown expression type -- EVAL") (display exp) (newline))))

;;; eval-definition
(define (eval-definition exp env)
  (define-variable!
    (definition-variable exp)
    (myeval (definition-value exp) env)
    env)
  'ok)

(define (definition-variable exp)
  (if (symbol? (cadr exp))
      (cadr exp) ; variable  e.g. (define x 2)
      (caadr exp))) ; procedure   e.g. (define (f x) (+ x 1))
```

```
(define (definition-value exp)
  (if (symbol? (cadr exp))
      (caddr exp) ; variable
      (cons 'lambda (cons (cdr (car (cdr exp)))
                          (cdr (cdr exp))))))  ; procedure

;;; eval-if
(define (if-pred exp) (cadr exp))
(define (if-then exp) (caddr exp))
(define (if-else exp)
  (if (not (null? (cdddr exp)))
      (cadddr exp)
      #f))
(define (eval-if exp env)
  (if (myeval (if-pred exp) env)
      (myeval (if-then exp) env)
      (myeval (if-else exp) env)))

;;; eval-lambda  e.g. (lambda (x y) (+ x y))
(define (eval-lambda exp env)
  (list 'procedure
        (car (cdr exp)) ; lambda-parameters
        (cdr (cdr exp)) ; lambda-body
        env))

;;; eval-application
(define (eval-application exp env)
  (let ((evaled-exp (map (lambda (x) (myeval x env)) exp)))
    (myapply (car evaled-exp) (cdr evaled-exp))))

;;; apply
(define (eval-seq exps env)
  (cond ((null? (cdr exps)) (myeval (car exps) env))
        (else (myeval (car exps) env)
```

```
              (eval-seq (cdr exps) env))))


(define (myapply procedure arguments)
  (cond ((tagged-list? procedure 'primitive) (apply (cadr procedure) arguments))
        ((tagged-list? procedure 'procedure)
         (let ((procedure-parameters (car (cdr procedure)))
               (procedure-body (car (cdr (cdr procedure))))
               (procedure-environment (car (cdr (cdr (cdr procedure))))))
           (eval-seq
            procedure-body
            (extend-environment procedure-parameters arguments procedure-environment))))
        (else (display "Unknown procedure type -- APPLY") (display procedure) (newline))))

;;; environment
(define (make-frame variables values) (cons variables values))
(define (frame-variables frame) (car frame))
(define (frame-values frame) (cdr frame))
(define (add-binding frame var val)
  (set-car! frame (cons var (frame-variables frame)))
  (set-cdr! frame (cons val (frame-values frame))))

(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      (if (< (length vars) (length vals))
          (begin (display "Too many arguments! ") (display vars) (display " ") (display vals)
(newline))
          (begin (display "Too few arguments! ") (display vars) (display " ") (display vals)
(newline)))))

;;; lookup-variable-value
(define (lookup-variable-value var env)
  (define (scan vars vals)
    (cond ((null? vars) '())
          ((eq? (car vars) var) (list (car vals)))
```

```
          (else (scan (cdr vars) (cdr vals)))))
    (define (env-loop env)
      (if (eq? env '())
          (begin (display "Unbound variable") (display var) (newline))
          (let ((v (scan (frame-variables (car env))
                         (frame-values (car env)))))
            (if (null? v)
                (env-loop (cdr env))
                (car v)))))
    (env-loop env))

; define-variable!
(define (define-variable! var val env)
  (let ((frame (car env)))
    (define (scan vars vals)
      (cond ((null? vars) (add-binding frame var val))
            ((eq? var (car vars)) (set-car! vals val))
            (else (scan (cdr vars) (cdr vals)))))
    (scan (frame-variables frame) (frame-values frame))))

; setup-environment
(define (setup-environment)
  (let ((initial-env
         (cons (make-frame (list 'car 'cdr 'cons 'null? '+ '- '* '/ 'remainder '< '> '<= '>=)
                           (map (lambda (x) (list 'primitive x))
                                (list car cdr cons null? + - * / remainder < > <= >=)))
               '())))
    (define-variable! 'true #t initial-env)
    (define-variable! 'false #f initial-env)
    initial-env))
```