「プログラミング言語」
SICP 第4章
～超言語的抽象～
その2

五十嵐 淳

igarashi@kuis.kyoto-u.ac.jp

京都大学

June 19, 2013

今日のメニュー

- **先週の宿題**

4.1 The Metacircular Evaluator

4.1.5 Data as Programs

4.1.6 Internal Definitions

4.1.7 Separating Syntactic Analysis from Execution

Ex 4.4 (and についてだけ)

```
;; eval の変更は省略
(define (eval-and exps env)
  (cond ((null? exps) #t)
        ((last-exp? exps) (eval (first-exp exps) env))
        ((true? (eval (first-exp exps) env))
         (eval-and (rest-exps exps) env))
        (else #f)))

;; derived expression としての実装
(define (and->if exps)
  (cond ((null? exps) 'true)
        ((last-exp? exps) (first-exp exps))
        (else (list
               'if (first-exp exps)
               (and->if (rest-exps exps)) 'false))))
```

Ex. 4.11

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan var-vals)
      (cond ((null? var-vals)
             (env-loop (enclosing-environment env)))
            ((eq? var (caar var-vals))
             (cdar var-vals))
            (else (scan (cdr var-vals)))))
    (if (eq? env the-empty-environment)
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
          (scan frame))))
  (env-loop env))
```

```
(define (make-frame variables values)
  (define (zip vars vals)
    (if (null? vars) '()
        (cons (cons (car vars) (car vals))
              (zip (cdr vars) (cdr vals)))))
  (cons '(*frame*) (zip variables values)))

(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan var-vals)
      (cond ((eq? var (caar var-vals))
             (set-cdr! (car var-vals) val))
            ((null? (cdr var-vals))
             (set-cdr! var-vals (list (cons var val))))
            (else (scan (cdr var-vals)))))
    (scan frame)))
```

4.1.5 プログラムとしてのデータ

- プログラム = 特定のタスクをこなす機械
- 評価器 = 万能機械
  - 入力: 機械の記述 (Lisp プログラム)
  - その機械の動作を模倣
- しかも (それなりに) 単純なプログラムで書ける!
- `(* x x)` はプログラム？リスト？
- 組込みの `eval` 関数について

4.1.6 内部定義

関数本体内部の define の扱いについて

```
(define (f x)
  (define (foo y) ... (bar ...) ...)
  (define (bar z) ... (foo ...) ...)
  ...)
```

- 意図: foo と bar を同時に定義

  ⇒ 変数参照 bar, foo は内部定義を指すべき

- この評価器実装では foo, bar を逐次処理

  ⇒ 定義されるものが関数 (すぐには変数参照を伴わない) ならうまくいく

  ∵ 関数が呼ばれる時には両方とも定義がされている!

jakld と 4.1 の評価器の動作が食い違う例

```
(define (bar x) (cons x 1))
(define (f x)
  (define foo (bar x))
  (define (bar z) (cons z 2))
  foo)
(f 5)
```

- jakld: **エラー**
- **実装した評価器**: `(5 . 1)` ⇐ **これは変**!

「同時に定義」の定義

```
(lambda ⟨vars⟩                (lambda ⟨vars⟩
  (define u ⟨e1⟩)               (let ((u '*unassigned*)
  (define v ⟨e2⟩)) ⇒                  (v '*unassigned*))
  ⟨e3⟩)                           (set! u ⟨e1⟩)
                                  (set! v ⟨e2⟩)
                                  ⟨e3⟩)))
```

と読み替え．ただし，`'*unassigened*`**は変数参照の結果になるとエラーを起こす特別なシンボル．**

宣言 (環境への追加) は同時/初期化は逐次

```
(lambda ⟨vars⟩
  (let ((u '*unassigned*)
        (v '*unassigned*))
    (set! u ⟨e1⟩)
    (set! v ⟨e2⟩)
    ⟨e3⟩))
```

- e_1, e_2 中の u, v は正しい宣言を指す
- e_1 から値を得るための計算では u, v を参照してはいけない
- e_2 から値を得るための計算では v を参照してはいけない

Exercise 4.16

In this exercise we implement the method just described for interpreting internal definitions. We assume that the evaluator supports `let` (see exercise 4.6).

a. Change `lookup-variable-value` (section 4.1.3) to signal an error if the value it finds is the symbol `*unassigned*`.

b. Write a procedure `scan-out-defines` that takes a procedure body and returns an equivalent one that has no internal definitions, by making the transformation described above.

c. Install `scan-out-defines` in the interpreter, either in `make-procedure` or in `procedure-body` (see section 4.1.3). Which place is better? Why?

4.1.7 構文解析と実行の分離

- 今の評価器は構文解析とそれ以外の計算を交互に実行
  - 構文解析: 入力式の形による場合分け
  - それ以外の計算: 環境の操作，プリミティブの実行
- 同じ式を何度も評価すると非効率

$\Rightarrow$ **構文解析と計算を分離**し，**構文解析は一度だけ**行うような `eval`

- アイデア (1): **カリー化**
- アイデア (2): **先にできる計算の括出し**

アイデア(1): 関数のカリー化

カリー化 (Currying)

二引数関数を「最初の引数をもらったら『次の引数をもらって値を返す』関数を返す」関数として表現

```
(define (mult x)
  (lambda (y) (* x y)))
(define double (mult 2))   ;; 二倍する関数
(double 3)
 ⇒ 6
(double 6)
 ⇒ 12
```

アイデア(2): 先にできる計算の括出し

例: カリー化した指数関数

```
(define (pow n) (lambda (m) ;; m^n の計算
  (if (= n 0) 1
      (* m ((pow (- n 1)) m)))))
```

再帰呼び出しは m が与えられるまで発生しない
$\Longrightarrow$ n だけから計算できる部分(比較，条件分岐，再帰)を (lambda (m) ...) の外に括り出す

```
(define (pow n)
  (if (= n 0) (lambda (m) 1)
      (let ((p (pow (- n 1))))
        (lambda (m) (* m (p m))))))
```

analyze: カリー化と括出しを施した eval

```
(define (eval exp env)
  ((analyze exp) env))
(define (analyze exp)
  ;; 秘密は各 analyze-XXX 関数に…
  (cond ((self-evaluating? exp)
         (analyze-self-evaluating exp))
        ...
        ((application? exp)
         (analyze-application exp))
        (else ...)))
```

analyze-XXX 関数群 (1/4)

- 環境を受け取り値を返す関数を返す
- 先に/後に計算する部分の区別
  - eval-XXX の定義と比べてみよう

```
(define (analyze-self-evaluating exp)
  (lambda (env) exp))
(define (analyze-quoted exp)
  (let ((qval (text-of-quotation exp)))
    (lambda (env) qval)))
(define (analyze-variable exp)
  (lambda (env)
    (lookup-variable-value exp env)))
```

analyze-XXX 関数群 (2/4)

```
(define (analyze-if exp)
  ;; 部分式の解析は先にやる
  (let ((pproc (analyze (if-predicate exp)))
        (cproc (analyze (if-consequent exp)))
        (aproc (analyze (if-alternative exp))))
    (lambda (env)
      ;; 条件判断は後
      (if (true? (pproc env))
          (cproc env)
          (aproc env)))))
```

analyze-XXX 関数群 (3/4)

```
(define (analyze-sequence exps)
  (define (sequentially proc1 proc2)
    (lambda (env) (proc1 env) (proc2 env)))
  (define (loop first-proc rest-procs)
    (if (null? rest-procs) first-proc
        (loop (sequentially
                  first-proc (car rest-procs))
              (cdr rest-procs))))
  (let ((procs (map analyze exps)))
    (if (null? procs)
        (error "Empty sequence -- ANALYZE"))
    (loop (car procs) (cdr procs))))
```

analyze-XXX 関数群 (4/4)

```
(define (analyze-application exp)
  (let ((fproc (analyze (operator exp)))
        (aprocs (map analyze (operands exp))))
    (lambda (env)
      ;; apply 相当の処理
      (execute-application (fproc env)
        (map (lambda (aproc) (aproc env))
             aprocs)))))
```

apply相当の処理

```
(define (my-apply proc args)
  (cond ((primitive-procedure? proc) ...)
        ((compound-procedure? proc)
         (eval-sequence
           (procedure-body proc)
           (extend-environment
             (procedure-parameters proc)
             args
             (procedure-environment proc))))
    (else
     (error ...))))
```

apply 相当の処理

```
(define (execute-application proc args)
  (cond ((primitive-procedure? proc) ...)
        ((compound-procedure? proc)
         (;; 本体は Scheme 関数なので直接呼出可
          (procedure-body proc)
          (extend-environment
            (procedure-parameters proc)
            args
            (procedure-environment proc))))
    (else
     (error ...))))
```

残り (4.1.2 節–4.1.4 節) のコードは同じでよい

宿題： 6/26(水) 午前8時 締切

- Ex. 4.6, 4.16, 4.23
- **レポートには**
  - **考え方の説明**
  - **プログラムリストと考え方の対応**
  - **実行例**

  を示すこと
- **レポート** (pdf) **とプログラムファイルを提出システムを通じて提出**
- **友達に教えてもらったら、その人の名前を明記**
- web **は出典を明記** (**「同じ」回答は減点**)

問 1-1

- 考え方: 座標，向きを局所状態として持ち，座標・向きを操作する手続きを持つオブジェクトを作る．
- ポイント:
  - 局所状態の実現方法
  - `make-turtle` がオブジェクトそのものでなくオブジェクト生成手続きになっているか

解答例

```
(define (make-turtle)  ;; ここの括弧は重要
  (let ((x 0) (y 0) (orientation 0)) ;; 局所状態
    (define (forward n)
      (set! x (+ x (* n (cos orientation))))
      (set! y (+ y (* n (sin orientation)))))
    (define (turn d)
      (set! orientation (+ orientation d)))
    (define (dispatch m)
      (cond ((eq? m 'fwd) forward)
            ((eq? m 'turn) turn)
            ((eq? m 'where) (list x y))))
    dispatch))
```

問 1-2

ポイント:

- `make-title`, `t` **の指す先が手続きを示す** ⊙⊙ **になっているか？**
- `t` **の指す環境が亀の状態を表す局所環境を指しているか**
- **その局所環境が大域環境を指しているか**
- **局所環境に** `fwd` **などの手続きも含まれているか (難易度大)**
- **局所環境が手続きを含むもの，亀の状態を含むものの二段構えになっているか (難易度最大，実装方法による)**

解答例

問2

ポイント

- **順序だてて考える**
- **リストの** box-and-pointer **表記**
- **箱を無闇にコピーしない (箱は** `cons` **の回数だけしかできない)**
- `set-cdr!` **の意味**
  - `set!` **と違い第一引数が変数ではない場合もある**

解答

問3-1: 解答

(1) (2 5)

(2) (2 5 7)

(3) (2 4 5 7 4)

- (2 4 5 4 7) **ではない!**

問 3-2:

nums を (数の) リストとすると，(bar! nums) は 'done を返し，状態変化で nums の指すリストは元のリストを昇順にソートしたものになっている．

- **foo! の挙動**
  - **nums の要素数が 1 以下ならそこで終了**
  - **「先頭 $\leq$ 2 番目」でもそこで即終了**
  - **「先頭 $>$ 2 番目」なら入れ替えて，後続リストについて再帰**

  nums の 2 番目以降が昇順ソート済なら，foo! の結果 nums 全体が昇順ソート済になる
- **bar! は後ろの要素から順に foo! を呼び出す**

問 4: 解答例

```
(define (expand n m)
  (cons-stream (quotient (* 10 n) m)
    (expand (remainder (* 10 n) m) m)))
```

- **筆算の要領**
- **n * 10 を m で割った商が先頭要素**
- **後続は余りを使って同じことを続ければよい**