

「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 17, 2014

今日のメニュー

4.1 The Metacircular Evaluator

4.1.5 Data as Programs

4.1.6 Internal Definitions

4.1.7 Separating Syntactic Analysis from Execution

環境操作のインターフェース (復習)

- (lookup-variable-value \langle 変数 \rangle \langle 環境 \rangle)
⇒ \langle 環境 \rangle から \langle 変数 \rangle の値を取ってくる
- (extend-environment \langle 変数列 \rangle \langle 値列 \rangle \langle 環境 \rangle)
⇒ \langle 環境 \rangle に \langle 変数列 \rangle と \langle 値列 \rangle からなる新しいフレームを追加した新しい環境を返す
- (define-variable! \langle 変数 \rangle \langle 値 \rangle \langle 環境 \rangle)
⇒ \langle 環境 \rangle 中の最初のフレームに変数束縛を追加
- (set-variable-value! \langle 変数 \rangle \langle 値 \rangle \langle 環境 \rangle)
⇒ \langle 環境 \rangle 中の \langle 変数 \rangle の値を \langle 値 \rangle に更新

環境のデータ構造(復習)

ここでの(単純だがあまり効率のよくない)実装方法:

- 環境 = フレームのリスト
- フレーム = (同じ長さの)変数リストと値リストのペア

```
(define (make-frame variables values)
  (cons variables values))
(define (add-binding-to-frame! var val frame)
  ;; frame の先頭に束縛 var = val を追加
  (set-car! frame (cons var (car frame)))
  (set-cdr! frame (cons val (cdr frame))))
```

環境の拡張

- 変数列 vars, 値列 vals からなるフレームを追加
- ふたつの列の長さが等しいかチェック

```
(define (extend-environment vars vals base-env)
  (if (= (length vars) (length vals))
      (cons (make-frame vars vals) base-env)
      ... ;; エラー処理
  ))
```

変数の値の検索

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals) ...)
    (if (eq? env the-empty-environment)
        ;; 変数が最後まで見つからないならエラー
        (error "Unbound variable" var)
        (let ((frame (first-frame env)))
            ;; 最初のフレームを探索
            (scan (frame-variables frame)
                  (frame-values frame))))))
  (env-loop env))
```

変数の値の検索

```
(define (lookup-variable-value var env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond
        ((null? vars) ;; このフレームにはない
         (env-loop
          (enclosing-environment env)))
        ((eq? var (car vars)) ;; あった!
         (car vals))
        (else (scan (cdr vars) (cdr vals)))))
    (if (eq? env the-empty-environment)
        ...
        (env-loop env)))
```

変数の値の更新

lookup-variable-value とほぼ同じ構造

```
(define (set-variable-value! var val env)
  (define (env-loop env)
    (define (scan vars vals)
      (cond ...
            ((eq? var (car vars)) ;; あった!
             (set-car! vals val))
            ...))
    (if (eq? env the-empty-environment)
        ...
        (env-loop env)))
```


変数の定義

最初のフレームを見て

- 既に定義されていたら値を更新
- 未定義なら束縛を追加

```
(define (define-variable! var val env)
  (let ((frame (first-frame env)))
    (define (scan vars vals)
      (cond
        ((null? vars) ;; 未定義
         (add-binding-to-frame! var val frame))
        ((eq? var (car vars)) ;; あった!
         (set-car! vals val))
        (else (scan (cdr vars) (cdr vals)))))
      (scan (frame-variables frame)
            (frame-values frame))))
```

Exercise 4.11

Instead of representing a frame as a pair of lists, we can represent a frame as a list of bindings, where each binding is a name–value pair. Rewrite the environment operations to use this alternative representation.

- 教科書の実装: フレーム = 変数リストと値リストのペア
- この練習問題: フレーム = 変数とその値のペアのリスト

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?
- 組み込みの 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⟩  
  (define u ⟨e1⟩)  
  (define v ⟨e2⟩) ⇒  
  ⟨e3⟩)  
  
(lambda ⟨vars⟩  
  (let ((u '*unassigned*  
         (v '*unassigned*)))  
    (set! u ⟨e1⟩)  
    (set! v ⟨e2⟩)  
    ⟨e3⟩))
```

と読み替え。ただし、'*unassigned*' 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).

- Change `lookup-variable-value` (section 4.1.3) to signal an error if the value it finds is the symbol `*unassigned*`.
- 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.
- 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
 - ⇒ 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): カラー化
 - ▶ アイデア (2): 先にできる計算の括出し

アイデア (1): 関数のカーリー化

カーリー化 (Currying)

二引数関数を「最初の引数をもったら『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
⇒ 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： 6/24(火) 10:30 締切

- Ex. 4.11, 4.16
- 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なら、その人の名前を明記
- web は出典を明記 (「同じ」回答は減点)