工学部専門科目
「プログラミング言語」SICP
第3章〜Modularity, Objects, State〜
その1

五十嵐 淳

igarashi@kuis.kyoto-u.ac.jp

京都大学 大学院情報学研究科
通信情報システム専攻

April 15, 2014

今日のメニュー

ソフトウェアシステム構築のために知るべきこと

- **プログラミングの基本要素**
  - **複合的手続きの構成**
  - **複合的データの構成**
  - **抽象化 (まとまりに名前をつけて細部を忘れる)**
- **大規模プログラムの構成・設計手法**
  - “modularity” **が大事**

    they [large systems] can be divided “naturally” into coherent parts that can be separately developed and maintained.

  - **原理のひとつ: モデル化する対象の構造に基づいてプログラムの構造を決める**

3章で扱うふたつの「世界観」

- システム as 時間の経過に応じて挙動を変える「オブジェクト」の集まり
- システム as 情報の流れ (ストリーム) (3.5 節)

それぞれの世界観が必要とする言語機構

- 「状態変化」を扱うためのプログラミング機構
  - 代入に基づく計算から環境に基づく計算へ
  - identity の問題
- 「ストリーム」を扱うための機構
  - 対象システムの時間・計算機上の事象順序の分離
  - 遅延評価

3.1 破壊的代入と局所的状態

- システム as 局所的状態を持つオブジェクトの集まり
  - 状態を持つ…振舞がそれまでの履歴に依存する
  - 局所的…システムの他の部分から独立
- 状態を表す状態変数
- 破壊的代入 (assignment)…識別子 (名前) で状態を表すために必要な機構

3.1.1 局所状態変数

例: 銀行口座

- **預金引き出しのための関数** `withdraw`

```
;; 残高 100 ドルで開始
(withdraw 25)
75
(withdraw 25)

(withdraw 60)

(withdraw 15)
```

```
(define balance 100)  ;; 初期残高
(define (withdraw amount)
  (if (>= balance amount)
      (begin ;; 破壊的代入による残高の更新
        (set! balance (- balance amount))
        balance)
      "Insufficient funds"))
```

- **破壊的代入**: (set! ⟨変数⟩ ⟨式⟩)
  - ⟨変数⟩の値を⟨式⟩の値で更新
  - 式全体の値は不定 (処理系依存)
- **逐次実行**: (begin ⟨式$_1$⟩ ··· ⟨式$_n$⟩)
  - 左から順に評価して⟨式$_n$⟩の値を全体の値とする

残高を表す変数の局所化

```
(define new-withdraw
  (let ((balance 100))
    (lambda (amount)
      (if (>= balance amount)
          (begin
            (set! balance (- balance amount))
            balance)
          "Insufficient funds"))))
```

- `let` を使った局所化
- `let` と `lambda` の順序に注意!

```
(new-withdraw 25)

balance

(define balance 100)

(new-withdraw 30)
```

口座を生成する関数

```
(define (make-withdraw balance)
  (lambda (amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
        "Insufficient funds")))
```

- **呼び出し毎に「別の口座」を (表す関数を) 返す**

```
(define W1 (make-withdraw 100))
(define W2 (make-withdraw 100))
(W1 50)

(W2 70)

(W2 40)

(W1 40)
```

預金と引き出し

メッセージパッシング (2.4.3 参照) + 局所状態

```
(define (make-account balance)
  (define (withdraw amount)
    (if (>= balance amount)
        (begin
          (set! balance (- balance amount))
          balance)
        "Insufficient funds"))
  (define (deposit amount)
    (set! balance (+ balance amount))
    balance)
  (define (dispatch m) ...)
  dispatch)
```

預金と引き出し

メッセージパッシング (2.4.3 参照) + 局所状態

```
(define (make-account balance)
  (define (withdraw amount) ...)
  (define (deposit amount) ...)
  (define (dispatch m) ;; メッセージ処理
    (cond ((eq? m 'withdraw) withdraw)
          ((eq? m 'deposit) deposit)
          (else
           (error
             "Unknown request -- MAKE-ACCOUNT"
             m))))
  dispatch)
```

```
(define acc (make-account 100))
((acc 'withdraw) 50)

((acc 'withdraw) 60)

((acc 'deposit) 40)

((acc 'withdraw) 60)

(define acc2 (make-account 100))

((acc2 'withdraw) 30)
```

3.1.2 破壊的代入の導入のご利益

- うまく使えばモジュラーな設計に役立つ
- 例: 疑似乱数
  - 「バラバラ」な数列 $x_1, x_2, \ldots, x_n, \ldots$
  - 実は関数 f があって $x_{n+1} = f(x_n)$

```
(define random-init ...) ;; 初期値 x1
(define (rand-update x) ...)
  ;; 次の項を求める手続き(上の f)
(define rand
  (let ((x random-init)) ;; 現在の項を記憶する
    (lambda ()
      (set! x (rand-update x))
      x)))
```

「何で rand-update を直接使うんじゃだめなの？」

⇒ 最後に生成した乱数を**プログラム全体で**「連れ回さ」ないといけなくなる!

⇒ モジュラリティの低下を招く

モンテカルロ法のプログラムで確認してみよう!

モンテカルロ法

- 確率を使った定理
- 大規模なサンプリング (実際にサイコロを振る) 実験

を組み合わせて推論を行う
例:

Theorem (Cesàro の定理)

ランダムに選んだ整数ふたつが互いに素である確率は $6/\pi^2$

を使って π を求める.

状態を利用するバージョン

```
(define (monte-carlo trials experiment)
  (define (iter trials-remaining trials-passed)
    (cond ((= trials-remaining 0)
           (/ trials-passed trials))
          ((experiment)
           (iter (- trials-remaining 1)
                 (+ trials-passed 1)))
          (else
           (iter (- trials-remaining 1)
                 trials-passed))))
  (iter trials 0))
```

```
(define (estimate-pi trials)
  (sqrt
    (/ 6 (monte-carlo trials cesaro-test))))

(define (cesaro-test)
   (= (gcd (rand) (rand)) 1))
```

状態を利用しないバージョン

```
(define (estimate-pi trials)
 (sqrt
  (/ 6 (random-gcd-test trials random-init))))
```

```
(define (random-gcd-test trials initial-x)
 (define (iter remaining passed x)
   (let* ((x1 (rand-update x))
          (x2 (rand-update x1)))
    (cond
      ((= remaining 0) (/ passed trials))
      ((= (gcd x1 x2) 1) ;; cesaro-test 相当
       (iter (- remaining 1) (+ passed 1) x2))
      (else
       (iter (- remaining 1) passed x2))))))
  (iter trials 0 initial-x))
```

考察

- 乱数に関する処理の分離が困難!
  - ちなみに，教科書は `cesaro-test` まで分離し損なっているが，もう少しはマシにできる
- 局所的であるはず・べき状態が丸見え

もう少しマシなバージョン(by 五十嵐)

```
(define (cesaro-test x)
  ;; テスト結果と最後の乱数値をペアで返す
  (let* ((x1 (rand-update x))
         (x2 (rand-update x1)))
    (cons (= (gcd x1 x2) 1) x2)))
```

```
(define (monte-carlo trials experiment initx)
 (define (iter remaining passed x)
  (if (= remaining 0)
      (/ passed trials)
      (let ((result (experiment x)))
        (if (car result)
            (iter (- remaining 1)
                  (+ passed 1)
                  (cdr result))
            ...))))
  (iter trials 0 initx))
```

3.1.3 破壊的代入の導入の代償

- プログラム実行が単純な式の書き換えでは説明できない
  - 代入を (激しく) 使うプログラミング
    → 命令型 (imperative) プログラミング
  - 代入を使わないプログラミング
    → (純粋) 関数 (functional) プログラミング
- データの同値性とは何かの議論が難しくなる
- 物事を行う順序に気をつかう必要あり

計算 $\neq$ 単純な式の書き換え

1 章での手続呼出の説明: 代入 (substitution) モデル

```
(define (square x) (* x x))
(define (sum-of-square x y)
  (+ (square x) (square y)))
```

```
(sum-of-square 3 4)   ;; x に 3, y に 4 を代入
⟶ (+ (square 3) (square 4))
⟶ (+ (* 3 3) (square 4))
⟶ (+ 9 (square 4))
⟶ (+ 9 (* 4 4))
⟶ (+ 9 16)
⟶ 25
```

代入モデルのポイント

- 手続のパラメータは実引数を指す名前
- 名前が指す値は (有効範囲 (scope) 内で) どこでも同じ

破壊的代入のあるプログラム

```
(define (make-withdraw2 balance)
  (lambda (amount)
    (set! balance (- balance amount))
    balance))

((make-withdraw2 100) 25) ⟶
((lambda (amount)
    (set! balance (- 100 amount)) 100)
 25)
⟶ (set! balance (- 100 25)) 100
  ;; balance の値を更新して 100 を返す!?
```

代入モデル

- 手続のパラメータは実引数を指す名前
- 名前が指す値は (有効範囲 (scope) 内で) どこでも同じ

別のモデル (3.2 のトピック)

- 手続のパラメータは実引数を**格納した箱**を指す名前
- 名前が指す箱は (有効範囲 (scope) 内で) どこでも同じだが，箱の中身は *set!* で変わりうる
- 二種類の変数参照: 箱を指す/箱の中身を指す

「同じである」ことについて

make-withdraw2 の set! なしバージョン:

```
(define (make-decrementer balance)
  (lambda (amount) (- balance amount)))
```

Q: **以下で定義される D1, D2 は「同じ」か？**

```
(define D1 (make-decrementer 100))
(define D2 (make-decrementer 100))
```

Q: 以下で定義される W1, W2 は「同じ」か？

```
(define W1 (make-withdraw2 100))
(define W2 (make-withdraw2 100))
```

参照透過性

参照透過性 (referential transparency)

定義された名前を定義内容で置き換えても式の値が変わらないこと

`set!` **があると参照透過性が失なわれる．例えば，**

- `(+ (D1 20) (D1 20))` **と**
- `(+ ((make-decrementer 100) 20) (D1 20))`

からは同じ値を得るが，

- `(+ (W1 20) (W1 20))` **と**
- `(+ ((make-withdraw2 100) 20) (W1 20))`

からは違う値が出てくる (なぜ？)

「Peter と Paul には$100の預金がある」

- **状況の表現その1**

```
(define peter-acc (make-account 100))
(define paul-acc (make-account 100))
```

- **状況の表現その2**

```
(define peter-acc (make-account 100))
(define paul-acc peter-acc)
```

ふたつの違いはなにか？

エイリアシング (aliasing，**別名付け**)

ひとつの「もの」に別の(ふたつ以上の)名前が付くこと

identity, 状態変化,「同じである」こと

- 世の中の「もの」の多くについて，identity とその属性 (状態) の区別が大事
  - 例: 銀行口座とその預金残高
- 「同じである」とは (たいてい) identity が同じであることをいう
- 状態変化がない「もの」については，identity と属性を同一視して構わない $\Rightarrow$ 参照透過性

物事を行う順序について

functional な繰り返し版階乗関数

```
(define (factorial n)
  (define (iter product counter)
    (if (> counter n)
        product
        (iter (* counter product)
              (+ counter 1))))
  (iter 1 1))
```

imperative な繰り返し版階乗関数

```
(define (factorial n)
  (let ((product 1) (counter 1))
    (define (iter)
      (if (> counter n)
          product
          (begin
            (set! product (* counter product))
            (set! counter (+ counter 1))
              ;; この2行を交換すると…？
            (iter))))
    (iter)))
```

functional vs. imperative

imperative **なプログラミングでは**

- **処理の順序，すなわち状態変化の順序**
- **エイリアシング**

に気をつける必要あり

$\Rightarrow$ imperative **プログラミングは難しい!**

3.1 のまとめ

- 状態 (変数の値) の変化を引きおこす `set!`
- 局所状態をうまく使うと
  - 我々のまわりの「もの」のモデリングがしやすい
  - プログラムがすっきり (modular に) 書ける
- プログラムの実行モデルの複雑化
- 参照透過性の喪失
- エイリアシング，実行順に気をつける必要

3.2 節「評価の環境モデル」

予習ポイント:

- **環境** (environment)，**フレーム** (frame)，**束縛** (binding) **とは何か？どういう関係を持っているか？**
- **「環境モデル」において手続オブジェクト** (procedural object) **はどう表現されているか？**
- **手続呼び出し式の評価手順**

宿題： 4/22(火) 午前10:30 締切

- Ex. 3.1, 3.3, 3.7
- **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**
- **引用は出典を明記** (**「同じ」回答は減点**)