システム理論演習 II

(2) 続 関数的プログラミング

増原英彦 *(masuhara@graco.c.u-tokyo.ac.jp)*

1997 年 4 月 25 日

1 「抽象化」の復習

- 抽象化の第一歩は、値に名前を付けることである。

  プログラム中に同じ値が (同じ意味で) 使われていた場合、それを変数で置きかえる。

  例えば、

```
(+ (/ (* 3.14 10 10) (* 2 3.14 10))
   (/ (* 3.14 20.8 20.8) (* 2 3.14 20.8)))
```

  これが

```
(+ (/ (* pi radius-a radius-a) (* 2 pi radius-a))
   (/ (* pi radius-b radius-b) (* 2 pi radius-b)))
```

  のように抽象化される。

- 第二歩は、同じ計算手順に名前を付けることである。

  プログラム中で同じ計算パターンがあった場合、それを関数定義と関数呼出に置きかえる。

  上の例では、`(* pi □ □)` と `(* 2 pi △)` という計算パターンが二度出現しているので、

```
(define (area x) (* pi x x))
(define (circumference x) (* 2 pi x))

(+ (/ (area radius-a) (circumference radius-a))
   (/ (area radius-b) (circumference radisu-b)))
```

  のように置きかえ、さらに、`(/ (area ○) (circumference ○))` というものも二度出現しているので、

```
(define (area/circum x)
  (/ (area x) (circumference x)))

(+ (area/circum radius-a) (area/circum radius-b))
```

  となる。

  上の例のように、`radius-a` のような具体的な式だったところを、関数の引数に置き換えてゆくことを、パラメータ化すると言う。

2 高階関数: 関数のパラメータ化

ここまでは、ある演算の引数がパラメータ化される例を見ていた。実は、演算子の部分もパラメータ化できる。

例えば、(1) a から b までの数の和を求める関数 `sum-integers`、(2) a から b までの数を 2 乗した数の和を求める関数 `sum-cubes` と、(3) $\pi/8$ の級数展開

$$\frac{1}{1\cdot 3}+\frac{1}{5\cdot 7}+\frac{1}{9\cdot 11}+\cdots$$

を求める関数 `pi-sum` は、それぞれ下のような定義ができる。

```
;;; (1)
(define (sum-integers a b)
  (if (> a b)
      0
      (+ a (sum-integers (+ a 1) b))))
;;; (2)
(define (sum-cubes a b)
  (if (> a b)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))
;;; (3)
(define (pi-sum a b)
  (if (> a b)
      0
      (+ (/ 1.0 (* a (+ a 2))) (pi-sum (+ a 4) b))))
```

3 つの関数の定義をよく見ると、共通部分が多いことが分かる。つまり、どれも

```
(define (⟨名前⟩ a b)
  (if (> a b)
      0
      (+ ⟨項⟩ (⟨名前⟩ ⟨次⟩ b))))
```

という格好をしている。

このような計算パターンを抽象化できるとうれしい。Scheme では、このような抽象化は ⟨項⟩ や ⟨次⟩ の部分を関数の引数にした関数を作り、 各項や次の係数を与える関数を引数として呼び出すようにすればよい。

実際、抽象化された関数 `sum` は

```
(define (sum term next a b)
  (if (> a b)
      0
      (+ (term a)
         (sum term next (next a) b))))
```

のように定義される。これを使って先の 3 つの関数は、

```
;;; 補助関数
(define (identity x) x)
(define (inc x) (+ x 1))
;;; (1)
(define (sum-integers a b)
  (sum identity inc a b))
;;; (2)
(define (sum-cubes a b)
  (sum cube inc a b))
;;; (3)
(define (pi-sum a b)
  (define (pi-term x)
    (+ (/ 1.0 (* x (+ x 2)))))
  (define (pi-next x) (+ x 4))
  (sum pi-term pi-next a b))
```

のようになる。

このように、関数を引数として受け取る関数のことを、高階関数 (higher-order functions) と言う。

(ここで、 `pi-term` や `pi-next` が `pi-sum` の内側で定義されているのに、 `identity` や `inc` のような補助関数が外側で定義されているのは、それぞれがどれだけ一般的かによって決めている。)

課題 2-1: (`product`)

区間 $[a, b]$ の関数 f の積を計算する高階関数 `product` を `sum` にならって定義せよ。

さらに、 `product` を使って階乗を求める関数 `factorial` を定義し直せ。

課題 2-2: (`accumulate`)

関数 `sum` と `product` は、 a から b までの関数 f の値を加算や乗算で集めるという点で共通している。そこで、これらを抽象化した高階関数 `accumulate` を定義せよ。この関数は

```
(accumulate combiner null-value term next a b)
```

のように 5 引数の関数であり、 `combiner` が数の集め方 (加算や乗算)、 `null-value` が範囲外に来たときの値 (加算だったら `0`、乗算だったら `1`) であるとする。

さらに、 `sum`, `product` を `accumulate` を使って定義し直せ。

3 匿名の関数

`pi-sum` の例で見たように、関数パラメータを使うと、その場限りの関数が沢山出現する。こういった関数を、関数が必要としている場所に式として書く `lambda` 形式というものがある。

(lambda (⟨仮引数⟩ ...) ⟨本体の式⟩)

例えば、`pi-sum` の場合は

```
(define (pi-sum a b)
  (sum (lambda (x) (+ (/ 1.0 (* x (+ x 2)))))
       (lambda (x) (+ x 4))
       a b))
```

のように書ける。

4 局所変数と let

$$f(x, y) = x(1 + xy)^2 + y(1 - y) + (1 + xy)(1 - y)$$

のような式は、補助的な変数を使って

$$\begin{aligned} a &= 1 + xy \\ b &= 1 - y \\ f(x, y) &= xa^2 + yb + ab \end{aligned}$$

のように簡単化できる。

関数の引数は、上の a, b のような補助的な変数の目的にも使える。

```
(define (f x y)
  (define (f-helper a b)
    (+ (* x (square a))
       (* y b)
       (* a b)))
  (f-helper (+ 1 (* x y)) (- 1 y)))
```

さらに、この f-helper はその場限りの関数であるから、 `lambda` を使って、

```
(define (f x y)
 ((lambda (a b)
    (+ (* x (square a))
       (* y b)
       (* a b)))
  (+ 1 (* x y)) (- 1 y)))
```

ともできる。このような補助変数は、多くの場面で役に立つ。そのため、省略記法として `let` という形式があり、

```
(define (f x y)
 (let ((a (+ 1 (* x y)))
       (b (- 1 y)))
  (+ (* x (square a))
     (* y b)
     (* a b))))
```

のように使うことができる。

この `let` 形式は、

```
(let ((⟨変数 1⟩ ⟨式 1⟩)
      (⟨変数 2⟩ ⟨式 2⟩)
            ⋮
      (⟨変数 n⟩ ⟨式 n⟩))
  ⟨本体⟩)
```

であり、直感的な意味は「⟨式 1⟩ ~ ⟨式 n⟩ を評価して、その値を ⟨変数 1⟩ ~ ⟨変数 n⟩ に結びつける。そうした上で ⟨本体⟩ を評価する」である。正確には、

```
((lambda (⟨変数 1⟩ ... ⟨変数 n⟩)
   ⟨本体⟩)
 ⟨式 1⟩ ... ⟨式 n⟩)
```

と書いたのと同じことである。

課題 2-3: (変数のスコープ)

次のような関数 `foo` があったとき、`(foo 1)` の値はいくつになるか。また、その理由を説明せよ。

```
(define (foo a)
  (let ((a (+ a a))
        (b (+ a a)))
    (+ a b)))
```

5 高階関数の例

方程式の解を求める例を 2 つ挙げる。

5.1 区分法 (二分法)

関数 $f(x) = 0$ の解を求める方法として、

$f(x)=0$ の解が、 a と b の間にあり、 $f(a)<0<f(b)$ だったとする。このとき、 $f\left(\frac{a+b}{2}\right)>0$ ならば、 x は a と $\frac{a+b}{2}$ の間に、 $f\left(\frac{a+b}{2}\right)<0$ ならば、 x は b と $\frac{a+b}{2}$ の間にある。

という性質を使う方法がある。

これを Scheme の関数として定義すると、

```
(define (search f neg-point pos-point)
  (let ((midpoint (average neg-point pos-point)))
    (if (close-enough? neg-point pos-point)
        midpoint
        (let ((test-value (f midpoint)))
          (cond ((positive? test-value)
                 (search f neg-point midpoint))
                ((negatvie? test-value)
                 (search f midpoint pos-point))
                (else midpoint))))))
```

のようになる。

ただし、 (search f u v) の u, v は、 (fu) $<0<$ (fv) でなければならないので、次の関数を使うのが便利である。

```
(define (half-interval-method f a b)
  (let ((a-value (f a))
        (b-avlue (f b)))
    (cond ((and (negative? a-value) (positive? b-value))
           (search f a b))
          ((and (negative? b-value) (positive? a-value))
           (search f b a))
          (else
           (error "Values are not of opposite sign" a b)))))
```

これを使って、 $\sin(x)=0$ の $2<x<4$ にある解を求めるには、

```
(half-interval-method sin 2.0 4.0)
```

とすればよいし、 $1<x<2$ にある $x^3-2x-3=0$ の解を求めるには、

```
(half-interval-method
  (lambda (x) (- (cube x) (* 2 x) 3))
  1.0 2.0)
```

とすればよい。

5.2 不動点

方程式 $f(x)=x$ の解を f の不動点 (fixed-point) という。不動点は、適当な初期値 a から出発して、

$$a, f(a), f(f(a)), f(f(f(a))), \ldots, f(\cdots(f(a))\cdots), \ldots$$

のよう計算して (収束すれば) 求められる。

この不動点による方程式の解法は、以下のように定義できる。

```
(define (fixed-point f first-guess)
  (define (try guess)
    (let ((next (f guess)))
      (if (close-enough? guess next)
          next
          (try next))))
  (try first-guess))
```

例えば、 $\cos x=x$ の解は、

```
(fixed-point cos 1.0)
```

で求められる。また、 $y = \sin y + \cos y$ の解は、

```
(fixed-point (lambda (y) (+ (sin y) (cos y)))
             1.0)
```

さらに、平方根 $\sqrt{x} = y$ を求める問題に対しても、

$$y = \frac{1}{2}(y + x/y)$$

と変形して

```
(define (sqrt x)
  (fixed-point (lambda (y) (average y (/ x y)))
               1.0))
```

として求められる。($y = x/y$ だからといって (lambda (y) (/ x y)) としてはいけない数学が得意な人は理由を考えてみよう。)

5.3 返り値としての関数

いままで、関数への引数として関数を渡すというのを見てきたが、逆に、関数が結果の値として関数を返すこともできる。

先の sqrt の例のように、 $y = f(y)$ を解く場合に、

$$g(y) = \frac{1}{2}(y + f(y))$$

とおいて g の不動点を求めると解が求まりやすい (らしい)。これを average-damping という。

そこで、関数 f を受け取って関数 g を作るような関数を作る。

```
(define (avreage-damp f)
  (lambda (y) (average y (f y))))
```

これを使えば、

```
(define (sqrt x)
  (fixed-point (average-damp (lambda (y) (/ x y)))
               1.0))
```

のように定義できる。同様に、三乗根 $\sqrt[3]{x}$ は、

```
(define (cube-root x)
  (fixed-point (average-damp (lambda (y) (/ x (square y))))
               1.0))
```

のようになる。

このように、関数の結果として関数を作って返すことも可能である。

(C 言語の関数ポインタとの違いは何か?)

5.4 ニュートン法

ニュートン法とは、微分可能な関数 g の方程式、 $g(x) = 0$ を解く方法の一つで、

$$f(x) = x - \frac{g(x)}{\mathrm{D}g(x)}$$

なる関数 f の不動点を求めることに相当する。

ただし D は微分演算子で、充分小さい $\mathrm{d}x$ を使って

$$\mathrm{D}g(x) = \frac{g(x + \mathrm{d}x) - g(x)}{\mathrm{d}x}$$

で近似できる。

つまり、ニュートンを法で解くには、

- f から g を作る関数 `newton-transform`
- g の微分関数を $\mathrm{D}g$ を作る関数 `derive`

を定義してあげればよい。

```
(define dx 0.00001)
;;; g を微分する
(define (derive g)
  (lambda (x)
    (/ (- (g (+ x dx)) (g x))
       dx)))
;;; g から f を作る
(define (newton-transform g)
  (lambda (x)
    (- x (/ (g x) ((derive g) x)))))
;;; ニュートン法
(define (newtons-method g guess)
  (fixed-point (newton-transform g) guess))
;;; ニュートン法による平方根
(define (sqrt x)
  (newtons-method (lambda (y) (- (square y) x))
                  1.0))
```

6 まとめ

関数を引数として渡したり、関数を作って返したりできることによって、多くの計算がより抽象化できることを見てきた。このような性質を、関数が一級データ (first class data) であるとも言う。一般に、一級データであるということは、

- 変数によって名前が付けられる
- 関数の引数として渡すことができる
- 関数の返り値として受け取ることができる
- データ構造の中に入れることができる

といった性質を満たすものを言う。

課題 2-4: (比較)

Pascal の procedure, function は一級データか?

また、 C の関数ポインタは一級データか? もしそうなら、 Scheme の関数との違いはあるか?

課題 2-5: (`double`)

関数を引数として受け取り、「その関数を 2 回適用するような関数」を返す関数 `double` を定義せよ。つまり `((dobule f) x)` が `(f (f x))` となる。

そのときの `(((double (double double)) inc) 5)` の値は何か?

課題 2-6: (`compose`)

二つの関数 f, g を合成するような関数 `compose` を定義せよ。つまり、`(compose f g)` が `(f (g x))` となるようなものである。

そのときの `((compose square inc) 6)` の値は何か?

課題 2-7: (`repeat`)

関数 f と自然数 n を受け取り、「f を n 回適用するような関数」を返す関数 `repeat` を定義せよ。

(頭の体操) 関数 `repeat`, `inc` だけを使って和、積、冪乗を計算する関数をそれぞれ定義せよ。