# プログラミング入門
# 第５回「高階関数」

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

## プログラムの構造と解釈（第１章のつづき）

### 1.3 高階関数による抽象

関数に名前をつけ、関数の抽象化を行ってきました。Scheme では関数を引数として渡したり、関数を返値として受け取ることができ、これによってさらに関数を抽象化することができます。例えば、sort 関数は、どの順でデータを並べるか、データの大小を定義する関数を引数として渡すことができれば、数値以外のデータも並べ替えができるようになります。

#### 1.3.1 引数としての関数

次の関数 f の定義をみてみよう。

```
(define (f g x y z)
    (+ (g x) (g y) (g z)))
```

関数 f は g, x, y, z の４つの引数をもっていて、これらの引数を受け取ったあと、(+ (g x) (g y) (g z))の計算を行いその結果を返します。ただ、ここで、変に思うかもしれないところは、(g x), (g y), (g z)を評価している、ということです。Scheme では括弧の内側の一番左の要素が関数や演算子となっていました。つまり、ここをみると、g

は関数であることがわかります。g は f の引数でしたので、実は f は g を関数として受け取っている、ということがわかります。試しに(f square 2 3 4)を置き換えモデルで評価してみましょう。

(f square 2 3 4)

=(+ (square 2) (square 3) (square 4))

=(+ 4 9 16)

=29

このように f に対して 2, 3, 4 という数値だけでなく、square という関数も渡し、(+ (square 2) (square 3) (square 4))を計算している、ということになります。

f を定義したときには、f が具体的に何を計算するのかまだ定まっていなくて、f の評価時に関数 g を f に渡すことによって初めて具体的に何を計算するのか決まる、ということになります。つまり、f は抽象化された関数であり、g を渡すことによって初めて具体的に何を計算するのか決まる、というふうにもいえます。square を渡すだけでなく、他にも様々な関数を渡すことができます。例えば、square のかわりに cube を渡せば、次のように計算させることができます。

(f cube 2 3 4)

=(+ (cube 2) (cube 3) (cube 4))

=(+ 8 27 64)

=99

ただし、cube については、(define (cube x) (* x x x))とします。

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 function)と呼びます。高階関数を用いることにより関数の抽象化をさらに行うことができます。まず、次の二つの関数をみてみよう。

```
(define (sum-integers a b)
  (if (> a b)
      0
```

```
      (+ a (sum-integers (+ a 1) b))))

(define (sum-cubes a b)

  (if (> a b)

      0

      (+ (cube a) (sum-cubes (+ a 1) b))))
```

最初の関数は、a から b までの整数の和を計算し、次の関数は、a から b までの整数の三乗和を計算します。

これらの二つの関数は明らかに共通のパターンをもっていて、それは、次のように記述できます。

```
(define (⟨name⟩ a b)

  (if (> a b)

      0

      (+ (⟨f⟩ a)

         (⟨name⟩ (+ a 1) b))))
```

こういう共通のパターンがあれば、プログラムをより抽象化することができます。数学においても、例えば級数の和という非常に強力な抽象化を見いだし、総和記号が発明されました。

$$\sum_{n=a}^{b} f(n) = f(a) + \cdots + f(b)$$

と表記されるおなじみのこの表現です。

Schemeにおいても関数自身を引数として渡すことにより、同様の抽象化が行えます。

```
(define (sum f a b)

  (if (> a b)

      0
```

```
    (+ (f a)
      (sum f (+ a 1) b))))
```

これを用いれば例えば 1 から 10 までの三乗和は次のように書けます。

```
(sum cube 1 10)
```

1 から 10 までの整数和も次のように書けます。

```
(define (identity x) x)
(sum identity 1 10)
```

## 1.3.2 LAMBDA を使う関数の構築

高階関数を用いるのに引数で渡す関数を define でいちいち定義するのは非常に面倒ですし、プログラムが煩雑になってきます。そこで、関数を作り出す特殊形式 lambda(「ラムダ」と読みます)を用いれば、直接関数を作り出し、引数に渡すことができるようになります。

```
(lambda (x) (+ x 4))
```

と書けば、x という引数を持つ関数が返ってきます。上述の 1 から 10 までの三乗和や整数和は次のように書けます。

```
(sum (lambda (x) (* x x x)) 1 10) ; 1 から 10 までの三乗和
(sum (lambda (x) x) 1 10)) ; 1 から 10 までの整数和
```

lambda は一般に次のように使われます。

```
(lambda (⟨引数の列⟩) ⟨本体⟩)
```

lambda は関数に名前がつかないという点を除けば define で定義される関数とまったく同じです。唯一の相違は環境で名前と対応づけられていないことです。実際

```
(define (plus4 x) (+ x 4))
```

は

```
(define plus4 (lambda (x) (+ x 4))
```

と等価です。他に、例えば、

```
(define (f x y z) (+ x y (square z)))

(f 1 2 3)
```

と評価する代わりに、

```
((lambda (x y z) (+ x y (square z))) 1 2 3)
```

としてもまったく同じ結果となります。

## 局所変数を作り出す LET の使い方

関数の引数として与えられた変数以外にも局所変数が欲しくなることがあります。例えば、

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

を計算したくなったとします。すると、上の式を直接計算するよりも、明らかに$1+xy$と$1-y$を先に計算しておいて、それを局所変数に代入しておいて再利用するようにして計算するほうが効率が良さそうです。そこで、let という特殊形式を用います。

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

と書けます。let の一般形は

```
(let ((⟨var1⟩ ⟨exp1⟩)
      (⟨var2⟩ ⟨exp2⟩)
      ...
```

$(\langle var\ n\rangle\ \langle exp\ n\rangle))$

$\langle body\rangle)$

となります。これは$\langle body\rangle$の中で$\langle var1\rangle$は$\langle exp1\rangle$の値を持ち、$\langle var2\rangle$は$\langle exp2\rangle$の値を持ち、…、$\langle var\ n\rangle$は$\langle exp\ n\rangle$の値を持つとせよ、ということになります。

しかし、実は、define を使って同じ関数を書くことができます。

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

これと let を使った場合との違いを今説明するのは難しいのですが、let はみたとおりの有効範囲を細かくどこででも設定できますが、一方、define は設定できるところが限られていて、有効範囲がみたままとは限らない、という違いがあります。例えば、let は if 式や計算式の中でも書けますが、define はだめです。また、let は局所変数を細かく設定することができます。例えば、次のように記述することができます。

```
(+ (let ((x 3)) (+ x (* x 10)))
   x)
```

let の中の x は外にある x とは異なる変数として扱われます。

基本的には、内部関数は define で、その他の局所変数は let で記述するようにするとよいです。

## 1.3.4 値として返される関数

Scheme は関数を渡すだけでなく、関数を返すということもできます。次の式をみてみましょう。

((if (> 5 3) + -) 2 4)

まず、部分式を評価し、(if (> 5 3) + -)は+という関数を返し、次に(+ 2 4)が評価されるので、6 が返ってくる、ということになります。

((if (> 5 3) + -) 2 4)

=(+ 2 4)

=6

　次は関数を受け取り関数を返す導関数の例です。導関数は関数を受け取り、その関数を微分することにより得られます。

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

```
(define dx 0.00001)

(define (derive f)

  (lambda (x)

    (/ (- (f (+ x dx)) (f x))

       dx)))
```

とすれば、関数$f$を微分した$f'$が返されます。例えば、

```
(define (cube x) (* x x x))
```

とすると、

((derive cube) 5)

に対し、75.0001500001 が返ってきます。

```
(define dcube/dx (derive cube))
```

とすれば、dcube/dx という関数は cube の導関数（の近似）となり、普通の関数と同様に、

(dcube/dx 5)

とすると、上記と同じ結果の 75.0001500001 が返ってきます。

Lisp や Scheme は関数に完全な第一級身分を与えている、と言われています。効率的な実装が難しくなったかわりに非常に高い表現力を持っていることがわかります。