情報システム科学実習 II (4)

高階関数

1999 年 11 月 10 日 増原 英彦

1 関数のパラメータ化

関数を用いることで、同一の計算手順を一つの名前で表わすことを学んだ。その際、計算ごとに異なる値はパラメータとして与えていた。

多くの関数型言語では、関数をパラメータとして受け渡しできる。これによって大きな計算手順の中で、小さな計算だけが違うような場合に、その部分を関数パラメータとして抽象化できる。

例として以下の 4 つの関数を関数パラメータを使って抽象化することを見てみよう。

$$a + (a+1) + (a+2) + \cdots + b = \tag{1}$$

$$\texttt{sum-integers}(a,b) = \begin{cases} 0 & \text{if } b < a \\ a + \texttt{sum-integers}(a+1,b) & \text{otherwise} \end{cases} \tag{2}$$

$$a^3 + (a+1)^3 + (a+2)^3 + \cdots + b^3 = \tag{3}$$

$$\texttt{sum-cubes}(a,b) = \begin{cases} 0 & \text{if } b < a \\ a^3 + \texttt{sum-cubes}(a+1,b) & \text{otherwise} \end{cases} \tag{4}$$

$$a + (a+2) + (a+4) + \cdots + b = \tag{5}$$

$$\texttt{sum-by-two}(a,b) = \begin{cases} 0 & \text{if } b < a \\ a + \texttt{sum-by-two}(a+2,b) & \text{otherwise} \end{cases} \tag{6}$$

$$a \times (a+1) \times (a+2) \times \cdots \times b = \tag{7}$$

$$\texttt{products}(a,b) = \begin{cases} 1 & \text{if } b < a \\ a \times \texttt{products}(a+1,b) & \text{otherwise} \end{cases} \tag{8}$$

```
(define (sum-integers a b)      ; (2)
  (if (< b a)
      0
      (+ a (sum-integers (+ a 1) b))))

(define (sum-cubes a b)         ; (4)
  (if (< b a)
      0
      (+ (cube a) (sum-cubes (+ a 1) b))))

(define (sum-by-two a b)        ; (6)
  (if (< b a)
      0
      (+ a (sum-by-two (+ a 2) b))))

(define (products a b)          ; (8)
  (if (< b a)
      1
      (* a (products (+ a 1) b))))
```

(2) と (4) を比べると、足し合わせる項の形が違うだけで、他は同じである。そこで足し合わせる項の部分をパラメータ化して次のようにできる:

```
;; a から b までの整数に関数を適用したものの和
(define (sum-of-terms term a b)
  (if (< b a)
      0
      (+ (term a) (sum-of-terms term (+ a 1) b))))

(define (identity x) x)          ; 補助関数

(define (sum-integers a b) (sum-of-terms identity a b))
(define (sum-cubes a b) (sum-of-terms cube a b))
```

関数sum-integers の中では sum-of-terms に関数identity パラメータとして渡している。演算を行う場合とパラメータとして用いる場合では、括弧の有無が違うことに注意。一方、sum-of-terms の方ではterm という変数で受け取り、あたかもterm という名前の関数が定義されているかのようにして関数を呼び出している。

さらに、(6) と見比べると、a の増分が違っているだけであるので、ここをパラーメタ化する:

```
;; a から b までの整数に
(define (sum term next a b)
  (if (< b a)
      0
      (+ (term a) (sum term next (next a) b))))
(define (inc x) (+ x 1))         ; 補助関数
(define (add-two x) (+ x 2))     ; 補助関数
(define (sum-integers a b) (sum identity inc a b))
(define (sum-cubes a b) (sum cube inc a b))
(define (sum-of-odds a b) (sum identity add-two a b))
```

このように、(比較的) 複雑な繰り返しの定義を一つにまとめることで、

- プログラムを読む人が、繰り返しを使っている部分を認識しやすくなる
- 再利用性が高くなる
- 誤りを減らせる

といったメリットがある。

GUI 部品のように「何かが起きたら所定の動作をする」場合には、関数パラメータが非常に便利である。実はすでにボタンの例などですでに関数パラメータを使っている。例えば、

```
(make <Button> :text "Hello." :command greetings)
```

のようにボタンを作る場合には、ボタンがクリックされたときの動作を:command の後ろに関数パラメータとして渡している。

課題 **4-1:** (関数パラメータ)

(2) ~ (8) をまとめて関数パラメータ化した関数 accumulate を定義せよ。さらに (2) ~ (8) をaccumulate を使って再定義せよ。 ∎

課題 **4-2:** (文字列の変換)

1. 文字列中の全ての文字を文字にする関数は例えば次のように定義できる:

```
;; 文字から長さ 1 の文字列を作る
(define (character->string character)
  (make-string 1 character))

;; 文字列中の各文字を大文字にする
(define (upcase-string string)
  (let ((len (string-length string)))
    (if (= len 0)
        ""                          ; 空文字列の大文化は空文字列
        (let ((top-character (string-ref string 0))
              (rest-string (substring string 1 len)))
          (string-append
           ;; 先頭の文字を大文字化したものと
           (character->string (char-upcase top-character))
           ;; 2 文字目以降の文字列を大文字化したものを結合
           (upcase-string rest-string))))))
```

   この関数のうち大文字化する部分をパラメータ化して、「文字列の各文字について変換をする」関数(`convert-string converter string`) を作れ。

2. 文字列を小文字化する関数(`downcase-string string`) を作れ。

∎

課題 **4-3:** (暗号)

1. ジュリアス・シーザーが使用したと言われる暗号は、例えば「A」を「M」に「B」を「N」にと、アルファベットを 13 文字ずらすものであったと言われる。まず、アルファベットを`shift` 文字ずらす関数(`rotate-alphabet shift char`) を定義せよ。`(rotate-alphabet 12 #\a) => #\m`、`(rotate-alphabet 12 #\w) => #\i` となるはずである。

2. アルファベットを 13 文字ずらす(`caesar-encrypt-char char`) と、逆向きに 13 文字ずらす(`caesar-decrypt-char char`) を定義せよ。

3. 文字列中の各文字を 13 個ずらす(`caesar-encrypt string`) と、逆向きにずらす(`caesar-decrypt string`) を定義せよ。

4. “oazsdmfgxmfuaze” を解読せよ。

∎

2　名無しの関数

関数パラメータを用いる場合、(define (inc x) (+ x 1)) のように、一回限りでしか使われない関数を沢山定義する必要が生じる。こういった場合、lamba 形式によって、名前のない関数を定義できる。以下の式が評価されると、n 個の引数を受け取り、それぞれを変数 1 ~ 変数 n に結び付けて式を計算するような関数が作られる。

(lambda (変数 1 変数 2 ...変数 n) 式)

これを例いれば、前出のsum-integers は補助関数inc やidentity を使わずに定義できる。

```
;; term(a)+term(next(a))+...term(next^n(a)) を計算
;; ただし next^n(a) <= b < next^n+1(a)
(define (sum term next a b)
  (if (< b a)
      0
      (+ (term a) (sum term next (next a) b))))

;; a から b までの整数の和
(define (sum-integers a b)
  (sum (lambda (x) x) (lambda (x) (+ x 1)) a b))
```

余談:　これまで見てきたdefine による関数定義やlet、begin のような形式は全てlambda を使って書き換えることができる。以下にそれを示す。このように Scheme では多くの言語要素が少数の言語要素によって説明できるため、言語の意味定義やプログラムの解析、コンパイラの作成などを容易にしている。

```
関数定義は、名無しの関数を変数に結びつける
(define (f v1 v2 v3 ...) 式)
= (define f (lambda (v1 v2 v3 ....) 式)

関数は仮引数に値を束縛するので let による一時変数と同じ効果を持つ
(let ((v1 式 1) (v2 式 2) ...) 式)
= ((lambda (v1 v2 ... vn) 式)
   式 1 式 2 ... 式 n)

関数適用は引数が先に評価されるので、順番に計算することができる
(begin 式 1 式 2)
= ((lambda (dummy) 式 2)
   式 1)
```

3　関数クロージャ

関数の中でlambda 形式を使って関数を作る場合、外側の変数を使って計算することもできる。

```
(define (adder x) (lambda (y) (+ x y)))
(define add-3 (adder 3))
(define add-100 (adder 100))
(+ (add-3 1) (add-100 1))
```

上でadder は引数x を受け取り、「与えられた数にx を加えた値を計算する」関数を返す。例えば(adder 3) の答えは、「与えられた数に 3 を加えた値を計算する」関数である。このようにして作った関数をadd-3 とadd-100 という名前を与えやると、両者は同じ式を使っているが、その中におけるx の値はそれぞれ異なっている。

このようにlambda 形式によって作られた関数は、単に式だけでなく、その式の外側で定義されている変数の情報も持っている。このような性質を強調して関数クロージャ(function closure) と呼ぶこともある。

例: GUI のようなアプリケーションにおいても関数クロージャは有効である。以下の例では、

- messenger はメッセージを表示する関数クロージャを作る
- make-fortune-button の :command 部は、動作 (メッセージの表示) は同じだが、値 (メッセージの内容) だけが違うボタンを作る

といった目的で関数クロージャが使われている。(Web ページに定義あり。)

```
;; メッセージを表示する関数を作る関数
(define (messenger message)
  (lambda () (display message) (newline)))

;; 押されるとメッセージを表示するボタンを作る関数
(define (make-fortune-button button-name message)
  (make <Button>
        :text "try this"
        :command (messenger message)))

;; おみくじ
(define (fortune-panel)
  (pack
   (make-fortune-button "Your defeats will turn into victories.")
   (make-fortune-button "Love smiles upon you this week.")
   (make-fortune-button "Write a letter to an old friend.")))
```

カリー化・関数合成: 複数の引数をとる関数を、1 引数の関数の並びに変換することをカリー化(currying) と言う。例えば、(lambda (x y) (+ x y)) のような関数をカリー化すると(lambda (x) (lambda (y) (+ x y))) になる。

2 引数の関数をカリー化する関数を定義しよう:

```
(define (curry f)
  (lambda (x) (lambda (y) (f x y))))
```

この関数を使えば、複数の引数をとる関数のうち一部分にだけ値を与えることが容易にできる。例えば、「1 加える」関数(lambda (x) (+ x 1)) は((curry +) 1) のように定義できる。

複数の関数を合成して新しく関数を定義することもできる。つまり、関数 f, g に対して $(f \circ g)(x) = f(g(x))$ となるような関数 $f \circ g$ を作ることである。1 引数の関数どうしを合成する関数を定義しよう:

```
(define (compose f g)
  (lambda (x) (f (g x))))
```

これを使えば、「2倍して1を足す」関数を`(compose ((curry +) 1) ((curry *) 2))`のように定義できる。