情報システム科学実習II (3)

再帰によるくり返し

1999年10月27日 増原 英彦

補足事項:

- (第2回レジュメB.1) ファイル`~/.emacs`に書くべき内容を以下のように修正:

```
(setq scheme-program-name "/home/masuhara/lib/stk/bin/stk")
```

- 作ったボタンなどの消し方:`(define b (make <Button> :text "Hello"))`のようにして作ったボタンは`(destroy b)`のようにして消すことができる。STkウィンドウ中の全ての部品を消すには`(for-each destroy (children *top-root*))`とする。これは、トップレベルウィンドウに含まれる全ての部品に対して`destroy`という関数を適用するという意味である。
- デバッグ方法。
- レポートの締切。

1 帰納的定義と再帰

「何かを繰り返して計算したい」ということは多い。Cのような手続き的プログラミングではwhileやforのようなループによって書くことが多かったのだが、関数の中から自分自身を呼び出す再帰(recursion)によって計算する。

例として、$1 + 2 + \cdots + n$を求める関数を定義してみよう。この関数を$\mathrm{sum}(n)$と書くことにすると、次のように帰納的に定義できる。

$$\mathrm{sum}(n) = \begin{cases} 1 & n = 1\ \text{のとき} \\ n + \mathrm{sum}(n-1) & \text{それ以外} \end{cases}$$

この定義の中ではsum自身を使っていることに注意。これは、ほとんどそのままSchemeの定義に置き換えることができる:

```
(define (sum n)
  (if (= n 1)
      1                          ; nが1のとき
      (+ n (sum (- n 1)))))      ; それ以外
```

練習 3-1: (再帰)

次の計算をする関数を定義せよ。

1. n の階乗 $(n! = 1 \times 2 \times \cdots \times n)$ の帰納的な定義と Scheme の関数 `(factorial n)`
2. 「1 から n までの中にある m の倍数の個数」を求める Scheme の関数`(count-multiples n m)` を定義せよ。この関数を $f(n,m)$ とすると、帰納的には次のように定義できる。

$$f(n,m) = \begin{cases} 0 & n = 0 \text{ の場合} \\ 1 + f(n-1,m) & n \text{ が } m \text{ の倍数の場合} \\ f(n-1,m) & \text{それ以外} \end{cases}$$

3. ある文字がある文字列の中に中回出現しているかを数える Scheme の関数 `(count-characters-in-string str char)` を定義せよ。(ヒント: `(string-length s)` は文字列 `s` の長さを計算する。`(string-ref s i)` は文字列 `s` の `i` 番目の文字を計算する。)
4. 積分の台形公式は

$$\int_a^b f(x)\mathrm{d}x = \left(\frac{1}{2}f(a) + f(a+\Delta) + f(a+2\Delta) + \cdots + f(a+(n-1)\Delta) + \frac{1}{2}f(b)\right)\Delta$$

(ただし $\Delta = (b-a)/n$) である。これを用いて、$\int_a^b \sin x\mathrm{d}x$ を求める`(integral-sin a b n)` を定義し、$\int_0^\pi \sin x\mathrm{d}x$ を求めよ。

■

課題 3-1: (素数)

ある数 n が $2, 3, \ldots, n-1$ のどれによっても割り切れない場合、n は素数である。n が素数であるかどうかを判定する関数を以下の順に定義せよ。

1. ある数 `n` がある数 `m` で割り切れる場合に真、そうでない場合に偽となる `(divisible? n m)`。
2. n が l 以上 n 未満の整数でどれかで割り切れる場合に真、そうでない場合に偽となる `(exists-divisor-from? n l)`。この関数を $f(n,l)$ とすると、次のように定義できる:

$$f(n,l) = \begin{cases} \text{偽} & l \geq n \text{ の場合} \\ \text{真} & n \text{ が } l \text{ の倍数の場合} \\ f(n,l+1) & \text{それ以外} \end{cases}$$

(この定義は工夫すれば `cond` や `if` を用いずに`and` や`or` を用いて作れる。)

3. n が素数であるとは n が 2 以上 n 未満のいずれの整数でも割り切れない場合である。`(prime? n)` を`exists-divisor-from?`を利用して定義せよ。

■

2　再帰の応用: ニュートン法

ニュートン法は、(細い話は省略するが) 方程式 $f(x)=0$ の解を数値的に求める方法で、

$$\begin{aligned} x_0 &= \text{適当な初期値} \\ x_{i+1} &= f'(x_i)(x-x_i)+f(x_i) \end{aligned}$$

のようにして、x_i を求めてゆき、$f(x_i)$ が充分 0 に近くなったところを近似解とするものである。

例えば、n の平方根を求めたかったら、$f(x)=x^2-n,\ x_0=1$ とすればよいので

$$\begin{aligned} x_0 &= 1 \\ x_{i+1} &= \frac{1}{2}\left(x_i+\frac{n}{x_i}\right) \end{aligned}$$

とすればよい。

これを Scheme で書くと、

```
(define (sqrt-iter x n)
  (if (good-enough? x n)
      x
      (sqrt-iter (improve x n) n)))
```

のようになる。`(good-enough?  x n)` は、$x^2 \simeq n$ であるかを判定する関数である。ここでは $|x^2-n|<\delta$ で近似を終了することにして、

```
(define delta 0.00001)
(define (good-enough? x n)
  (< (abs (- (square x) n)) delta))
```

と書ける。また、`(improve x n)` は、x_i と n から x_{i+1} を求める関数で、

```
(define (improve x n)
  (average x (/ n x)))
```

ただし、`average` は二数の平均をとる関数で、

```
(define (average x y)
  (/ (+ x y) 2))
```

である。最終的に、平方根を求める関数は $x_0=1$ で始めることにしたので、

```
(define (sqrt n)
  (sqrt-iter 1.0 n))
```

となる。

3 くり返し

くり返し的な処理も関数の再帰呼び出しによって定義できる。この場合、次のような構造になることが多い。

```
(define (くり返し処理 パラメータ...)
  (if 処理を続けるか?
      (begin
        処理の中身
           ⋮
        (くり返し処理 (次のくり返しのパラメータを計算)))))
```

例として、「タートルグラフィクス」によって絵を描いてみよう。タートルグラフィクスでは、陸亀に対して「何歩進め」(`walk`) と「曲がれ」(`turn`) という指示が出せる。

```
(load "turtle.scm") ; web ページによって提供

;; タートルグラフィクスの領域を作り
(define gamera (make <Turtle> :width 300 :height 300))
;; 表示する
(pack gamera)

(walk gamera 50)                ; 50 歩進め!
(turn gamera 45)                ; 45 度まわれ!
(jump gamera 100)               ; 100 歩飛べ!

;; (reset gamera)               ; 画面を消して元に戻す
```

これを使って一辺の長さが l の正 n 角形を描くには、以下のように繰り返しをすればよい(web ページに定義あり):

```
;; regular-polygon.scm
;; count 個の辺を描く
(define (polygon-side turtle count side-length delta-angle)
  (if (< 0 count)
      (begin
        (walk turtle side-length)
        (turn turtle delta-angle)
        (update)                        ; 画面を更新
        (polygon-side turtle (- count 1)
                      side-length delta-angle))))

;; 正 num-edges 角形を描く
(define (regular-polygon turtle num-edges side-length)
  (polygon-side turtle num-edges side-length (/ 360 num-edges)))
```

4 複雑な再帰

ここまで見た再帰呼び出しでは、1 つの関数からその関数を 1 回だけ呼びしている。もっと複雑な再帰も同様に扱える。

例として木を描く関数を定義してみよう。ここでは、幹を描いたあと、次の段階の枝を 2 つ描くために自分自身を 2 回呼び出している。(Web ページに定義あり。)

```
;; tree.scm            (turtle.scm を使用)
(define minimum-branch-length 5) ; 最小の枝の長さ

;; 枝の長さ length, 枝の縮少率を length-factor,
;; 分岐の角度を angle として木を描く
(define (tree turtle length length-factor angle)
  (update)
  ;; 枝が充分長ければ計算を続ける
  (if (< minimum-branch-length length)
     (let ((new-length (* length-factor length))); 次の枝長
       (walk turtle length)                     ; 幹を描く
       (turn turtle angle)
       (tree turtle new-length length-factor angle); 次の枝 1
       (turn turtle (* -2 angle))
       (tree turtle new-length length-factor angle); 次の枝 2
       (turn turtle angle)
       (jump turtle (- length)))))         ; 元の位置に戻る
```

課題 3-2: (フィボナッチ数)

フィボナッチ数列 (Fibonacci numbers) とは、

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

のように、「前 2 つの数の和」で作られる数の列である。n 番目のフィボナッチ数 $fib(n)$ を帰納的に定義せよ。また、Scheme 関数`(fibonacci n)` を定義せよ。

▮

課題 3-3: (タートルグラフィックス)

タートルグラフィックスライブラリを利用して、絵を描く関数を定義せよ。ただし、再帰を用いること。 ▮

5 おまけ: 関数のブラックボックス化:

- 問題を小問題に分割 → それぞれに関数定義
- 小問題はそれぞれブラックボックスとして扱える
- 細かな関数定義が沢山できてしまう

  → 局所的な定義によるブロック構造化

```
(define (sqrt n)
  (define delta 0.00001)
  (define (good-enough? x n)
    (< (abs (- (square x) n)) delta))
  (define (improve x n)
    (average x (/ n x)))
  (define (sqrt-iter x n)
    (if (good-enough? x n)
        x
        (sqrt-iter (improve x n) n)))
  (sqrt-iter 1.0 n))
```

- 名前の見える範囲 (スコープ)

  ある関数定義の内側で定義されている名前は、その関数定義の中だけしか見えない。上のようなブロック化をした場合は、`delta`, `good-enough?`等の名前は`sqrt`の定義の内側でしか使えない。

  一方、関数の内側で定義している関数には、その外側の名前が見えている。例えば、`improve`の定義内部では、`sqrt`への引数`n`が見えている。そのため、上の定義はもっと簡略化できて、次のようにもできる。

```
(define (sqrt n)
  (define delta 0.00001)
  (define (good-enough? x)
    (< (abs (- (square x) n)) delta))
  (define (improve x)
    (average x (/ n x)))
  (define (sqrt-iter x)
    (if (good-enough? x)
        x
        (sqrt-iter (improve x))))
  (sqrt-iter 1.0))
```