計算機プログラミング I (7)

再帰的なメソッド定義

2002 年 11 月 28 日　増原 英彦

あるメソッドの定義の中で、そのメソッド自身が使われているものを再帰的 (**recursive**) なメソッド定義という。再帰的なメソッド定義を使うと、帰納的な考え方をそのままプログラムにできる。そのため、プログラムの働きや正しさを考えるのにも適した強力なプログラム方法になっている。

1　帰納的定義

計算や処理を定義する場合に、「最初の場合 (ベースケース)」と「前の定義から 1 つ進めた場合」の定義によって、あらゆる場合の定義をする方法を帰納的(inductive) 定義という。

例として x の y 乗を定義する。これは y に関する帰納的な定義ができる。ベースケースは $y = 0$ のときで、$x^0 = 1$ である。「1 つ進めた場合」は y が 1 増えた場合になるので、x^{y-1} が定義済みである[1]として x^y を定義することになり、$x^y = x \times x^{y-1}$ となる。これをまとめると、次のように書ける:

$$x^y = \begin{cases} 1 & (y = 0 \text{ の場合}) \\ x \times x^{y-1} & (y > 0 \text{ の場合}) \end{cases}$$

冪乗 (x^y) を定義するのに、冪乗自身 (x^{y-1}) を使っていることに注意せよ。

練習 **7-1:** (帰納的な定義)*　次のような計算の、y に関する帰納的な定義を書け。

- x の y 倍、つまり $x \times y$。(ヒント: $x \times (y-1)$ を使う)
- y の階乗 ($y! = y \times (y-1) \times (y-2) \times \cdots \times 3 \times 2 \times 1$)。(ヒント: $(y-1)!$ を使う)

1.1　より複雑な帰納的定義

帰納的な定義は、自分自身を 2 回以上使うこともある。フィボナッチ数列 (Fibonacci numbers) は、x 番目の数が $x-1$ 番目と $x-2$ 番目の数の和になっているような数列であり、$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$ のようになっている。$f(x)$ をフィボナッチ数列の x 番目の数だとすると、$f(x)$ は $f(x-1)$ と $f(x-2)$ を使って帰納的に定義できる。

また、2 つ以上のパラメータに関する帰納的な定義も可能である。例えば、第 5 回で説明した組み合わせ数 ${}_mC_n$ は m と n という 2 つのパラメータに関しての帰納的定義で、${}_{m-1}C_n$ および ${}_{m-1}C_{n-1}$ を使って定義できる。

練習 **7-2:** (より複雑な帰納的定義)

- フィボナッチ数を帰納的に定義せよ。(ヒント: ベースケースは $x = 0$ と $x = 1$ の場合になる。)
- 組み合わせ数を帰納的な定義せよ。(ヒント: ベースケースは $n = 0$ と $n = m$ の場合になる。)
- $a > b$ なる整数 a, b の最大公約数 $GCD(a, b)$ を帰納的に定義せよ。(ヒント: 第 4 回に説明したユークリッドの互除法から、a, b の最大公約数は $(b, a\%b)$ の最大公約数と等しい。ただし、$a\%b$ は a を b で割った余りだとする。ベースケースに注意せよ。)

[1] x^{y-1} の値は帰納法による証明の、帰納法の仮定に相当する。

2　再帰的なメソッド定義

(注: 再帰的でないメソッドの定義は第 4 回講義資料を参照せよ。)

帰納的な定義は、ほとんどそのまま再帰的なメソッド呼出を使って定義できる。例えば、x の y 乗を計算するメソッドは次のように定義できる:

```
1 public static int power(int x, int y) {
2   if (y==0) {
3     return 1;                 // ベースケース
4   } else {
5     return x*power(x,y-1);    // y>0 の場合:  帰納的
6   }
7 }
```

- 1 行目:以下がクラスメソッドの宣言であること (static)、返り値の型 (int)、メソッドの名前 (power)、引数の型 (int とint) と、引数の値がセットされる変数名 (x とy) を示している。
- 2 行目の if 文は、y が 0 であれば 3 行目を実行し、そうでない場合 (つまり 0 以上) は 5 行目を実行させる。
- 3 行目は帰納的定義のベースケースに相当し、1 を返している。これは $x^0 = 1$ という定義に対応する。
- 5 行目は帰納的な場合に相当し、power(x,y-1) という式でx のy-1 乗を計算し、それとx を掛けた値を返している。これは $x^y = x \times x^{y-1}$ という定義に対応する。このpower(x,y-1) という式は自分自身を呼出している[2]、再帰的な呼出しである。

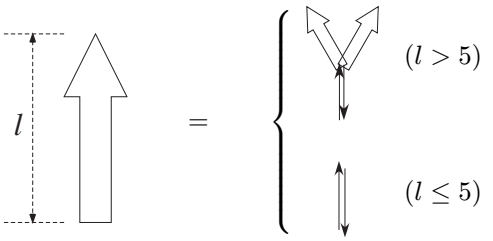
練習 **7-3:** (再帰的な冪乗メソッド)*　上のメソッド定義を用いて、2 の 0 乗から 30 乗までを計算し表示させるプログラムを作れ。余力があれば、練習 6-8 で示したメソッドと計算結果を比べ、同じ結果を返すことを確認せよ。

練習 **7-4:** (その他の再帰的メソッド)　練習 7-2 で作った帰納的な定義に従って、(a) フィボナッチ数、(b) 組み合わせ数、(c) 最大公約数、を計算するプログラムをそれぞれ作成せよ。さらに、組み合わせ数については練習 5-5 と、最大公約数については、練習 4-1,4-2 または 4-3 で作ったプログラムと同じ結果を返すことを確認せよ。

2.1　再帰的なメソッドとフラクタル図形

再帰的なメソッドは数学的な計算に限らない。木の枝は、中心となる太い枝に小さな枝が数本ついた形をしている。この小さな枝はまた、中心となる太い枝にさらに小さな枝が数本ついているという形をしている。つまり、木の枝は、太い枝に「木全体に相似な小さな枝」がついているといえる。

このような図形は、次のように帰納的に定義できる:



左辺の白い矢印を木だとすると、その長さ l が 5 より大きい場合は右辺上のように、幹に相当する線があり (実線の矢印)、その上端から左上と右上に向かって一まわり小さな木が伸びている (白い矢印)。幹の長さは $l/3$、

[2]クラスメソッドの呼出は クラス名.メソッド名(式, ...) のように書くが、同じクラスのメソッドの場合は単に メソッド名(式, ...) と書ける。

一まわり小さな木はそれぞれ左右に 30 度傾いて、長さが $2l/3$ だとする[3]。長さ l が 5 以下の場合は、右辺下の場合のように、単にその長さの線があるだけである。

この図形はタートルグラフィクスと次のような再帰的なメソッドによって描くことができる:

```
1  public static void drawTree(Turtle m, int l) {
2    if (l>5) {          // 長さが 5 より大きい場合は枝分かれする
3      m.fd(l/3);        // 幹を描く
4      m.lt(30);         // 左を向く
5      再帰的に 2/3 の長さの木 (枝) を描く
6      m.rt(60);         // 右を向く
7      再帰的に 2/3 の長さの木 (枝) を描く
8      m.lt(30);         // 正面を向き直す
9      m.bk(l/3);        // 最初の位置に戻る
10   } else {            // 長さが 5 以下の場合は線を一本描くだけ
11     m.fd(l);          // 幹を描く
12     m.bk(l);          // 最初の位置に戻る
13   }
14 }
```

このクラスメソッドdrawTree(m,l) は、タートルオブジェクトm を使って (m の現在位置から、m の向いている方向に向かって) 長さl の木を描いた後に、m を元の場所・向きに戻す。

幹の部分は 3 行目で描かれる。その先の小枝は 4, 6 行目でm を適切な方向へ向け、5, 7 行目のdrawTree メソッドの呼出しでそれぞれ描く。(drawTree メソッドは小枝を描いた後、元の場所・向きに戻るように作られていることに注意。) 最後に、8, 9 で、最初の向きと位置に戻っている。

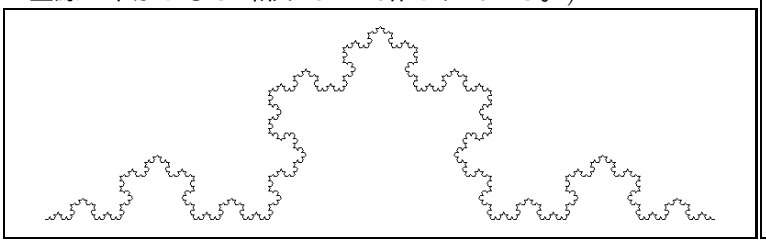
このような図形は自己相似形あるいはフラクタル図形といわれる。

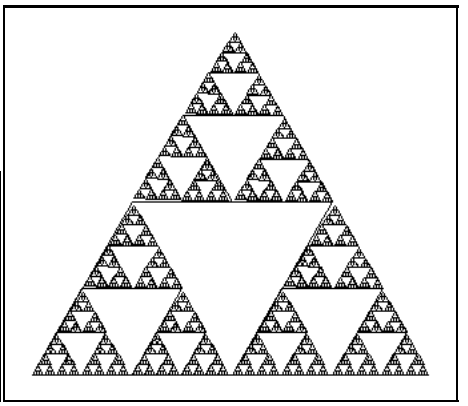
練習 **7-5:** (再帰的な木)* 上のメソッドを完成させ、実際に図形を描かせよ。(参考プログラム: Tree.java)

練習 **7-6:** (木のバリエーション) 上で作ったプログラムをもとに、(a) 枝分かれの本数を 2 本から 3 本にした木、(b) 枝分かれの本数・枝の開く角度・幹と枝の長さの割合、といった要素を乱数によって変更するプログラムを作り、より自然に見える木、をそれぞれ描くプログラムを作れ。

練習 **7-7:** (他のフラクタル図形)

右はシェルピンスキーの三角形、下はコッホ曲線と呼ばれるフラクタル図形である。これらの図形を描くプログラムを作成せよ。(ヒント: 一辺の長さが l のシェルピンスキーの三角形は、長さ $l/2$ の三角形 3 つで作られている。コッホ曲線は長さ l の直線が長さ $l/3$ の直線 4 本からなる山形によって作られている。)





3 再帰的メソッドの実行

drawTree(m,200) のようなメソッド呼出があると、その実行中に同じ名前のメソッドが呼出される。このとき、呼出す側のメソッドの実行と呼出される側のメソッドの実行は全く別のものとして扱われる。

より詳しく説明すると、メソッド呼出式が実行されるときはには、次のような処理が行われている。以下では区別のためにメソッド m1 がメソッド m2 を呼出すとして説明するが、同じ名前のメソッドを呼出したときでも同じである。

[3]この長さでは正確に長さ l の木を描くことにはならない。

1. 実行しようとしているメソッド呼出式が m1 の中どの場所にあるかを、また m1 のメソッドの中で宣言されている変数の値を覚えておく
2. メソッド m2 を実行 (し終了) する。
3. 覚えておいた変数の値を元に戻し、覚えておいた場所から m1 の実行を再開する

3 で覚えおいた場所と変数の値に戻るので、2 で同じ名前のメソッドを実行したとしても、呼出す側と呼出される側は全く別のものとして扱われる。

4 再帰的メソッドの応用

練習 7-8: (冪乗) 練習 6-8 で紹介した冪乗を計算する「別のアルゴリズム」は、再帰的メソッドを使って簡潔に定義できる。まず、x の n は次のように帰納的に定義できる:

$$x^n = \begin{cases} 1 & (n = 0) \\ (x^2)^{n/2} & (n \text{ は } 0 \text{ より大きい偶数}) \\ x \times x^{n-1} & (n \text{ は奇数}) \end{cases}$$

これをもとにメソッドを定義すると、以下のようになる:

```
1 public static int power(int x, int n) {
2     if (n==0) {
3         return 1;                  // n が 0 のとき
4     } else if (n%2 == 0) {
5         return (x^2)^{n/2}を計算;    // n が 0 より大きい偶数のとき
6     } else {
7         return x × x^{n-1} を計算; // n が奇数のとき
8     }
9 }
```

上のプログラムを完成させ、2 の 30 乗を計算させよ。また、メソッドの最初に`System.out.println(n);`という行を挿入し、メソッドが何回呼出されているかを調べよ。

練習 7-9: (ハノイの塔) 3 本の柱 (p_0, p_1, p_2 とする) と、n 枚の中央に穴の開いた円盤 ($d_0, d_1, \ldots, d_{n-1}$) があり、$d_i$ は d_{i+1} より小さいとする。初め全ての円盤は p_0 に下から $d_{n-1}, \ldots, d_0$ の順に差してあったとする。このとき、全ての円盤を p_0 から p_1 に移す手順を考えよ。ただし、1 つの柱に差してある円盤は 1 番上のものしか移動できず、より大きな円盤の上にしか重ねられないものとする。

p_f にある $d_i, \ldots, d_0$ を p_t へ移動する (ただし、残りの柱は p_e とする) には、次のような手順をふめばよい:

- p_f にある $d_{i-1}, \ldots, d_0$ を p_e へ移動し (ただし残りの柱は、p_t)、
- d_i を p_f から p_t へ移動し、
- p_e に移した $d_{i-1}, \ldots, d_0$ を p_t へ移動する (ただし残りの柱は、p_f)

これを再帰的なメソッドで定義すると、次のようになる。このプログラムを完成させよ。(参考 Hanoi.java)

```
1  /** 円盤 di ~ d0 を柱 pf から柱 pt に (柱 pe を使って) 移す */
2  public static void hanoi(int di, int pf, int pt, int pe) {
3    if (di == 0) {                          // ベースケース
4      di を pf から pt に移す
5    } else {                                // di の上に他の円盤があるとき
6      di-1 ~ d0 を pf から pe に (pt を使って) 移す
7      di を pf から pt に移す
8      di-1 ~ d0 を pe から pt に (pf を使って) 移す
9    }
10 }
```