プログラミング入門
第４回「再帰と反復」

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）

1.2 関数とその生成するプロセス

さて、関数の書き方はわかったし、if 文で場合分けすることもできるようになったし、局所変数の扱いもわかった。次は、計算の要である繰り返しや再帰呼び出しをどうやって実現するか、ということについて学びます。Scheme には、明示的な繰り返し構文はありません。かわりに末尾再帰（tail recursion）という再帰呼び出しを用います。

1.2.1 線形再帰と反復

$n!$（nの階乗）の計算について考えてみましょう。数学的関数では次のように定義できます。

$$factorial(n) = \begin{cases} 1 & if\ n = 1 \\ n \times factorial(n-1) & otherwise \end{cases}$$

これは、n が１なら１であり、そうでなければ、(n-1)の階乗を計算してから、n をかければよい、という再帰的な考えに基づいています。よくみると、factorial の中に factorial がまたでてきています。とりあえず、これを Scheme でそのまま実装してみましょう。次のようになります。

```
(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (- n 1)))))
```

このように factorial の中でもう一度 factorial を呼び出すことになります。このように、ある関数fがfの中で自分自身であるfを呼び出すことを再帰呼び出しといいます。また、再帰呼び出しには、再帰的プロセスと反復的プロセスの二種類の方法があります。

$$\text{再帰呼び出し}\begin{cases}\text{再帰的プロセス}\\ \text{反復的プロセス}\end{cases}$$

上の定義による階乗計算は再帰的プロセスと呼ばれる再帰呼び出しになります。置き換えモデルによる(factorial 6)の評価は次のようになります。(今後特に断らない限り全て作用的順序で評価します)

```
(factorial 6)
=(* 6 (factorial 5))
=(* 6 (* 5 (factorial 4)))
=(* 6 (* 5 (* 4 (factorial 3))))
=(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
=(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
=(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
=(* 6 (* 5 (* 4 (* 3 2))))
=(* 6 (* 5 (* 4 6)))
=(* 6 (* 5 24))
=(* 6 120)
=720
```

再帰的プロセス（recursive process)：この計算過程では、計算が遅延されていて、膨張し、収縮しています。このようなプロセスは再帰的プロセスと呼ばれます。このプロセスでは後で計算するべき演算を覚えておく必要があります。$n!$の場合は、覚えておくべき演算がnに比例して増えるため、線形再帰的プロセスと呼ばれます。

再帰呼び出しでは自分自身を呼び出しているため、if 文や cond 文を使って、「再帰呼び出しを止める工夫」をいれこまないと、無限ループに陥ってしまうことに気をつけましょう。再帰的プロセスでは数学的関数の定義、特に数学的帰納法の定義と相性がよいため、数学が得意な人にとってはプログラムを書きやすく、またバグもでにくい、ということになります。例えば、n の階乗の定義を書くときに、n の階乗は n-1 の階乗に n を書けたもの、ということは数学的に間違いないし、1 の階乗は 1 であることは数学的に間違いないわけです。つまり、必ず終了するように気をつけていれば、必ず正しい結果が得られることが数学的に保証されるわけです。

次に反復的プロセスについて説明します。上の方法以外にも、カウンターを用意して、カウンターを次々と掛け合わせることで、$n!$を計算する方法もあります。

```
(define (factorial n)
  (fact-iter 1 1 n))
(define (fact-iter product counter max-count)
  (if (> counter max-count)
      product
      (fact-iter (* counter product) (+ counter 1) max-count)))
```

この置き換えモデルによる評価は次のようになります。

```
(factorial 6)
=(fact-iter 1 1 6)
=(fact-iter 1 2 6)
=(fact-iter 2 3 6)
=(fact-iter 6 4 6)
```

=(fact-iter 24 5 6)

=(fact-iter 120 6 6)

=(fact-iter 720 7 6)

=720

反復的プロセス（iterative process): 再帰的プロセスの階乗計算では膨張、縮小をしていましたが、この方法では対照的に膨張、収縮をしていません。各引数にプロセスの状態を表す変数(状態変数, state variable)を渡し、状態を変更させていき、終了条件をみたしたときにその状態から計算結果を返すことができるようになっています。これを反復的プロセスと呼びます。$n!$の計算に必要なステップ数はnに比例するため、線形反復的プロセスと呼ばれます。反復的プロセスは上でわかるように固定長のスペースで計算できるため、末尾再帰的(tail recursive)とよばれます。この場合は、再帰呼び出しであっても、余分なスペースを必要としないため効率的に実行することができます(コンパイラやインタプリターが頑張るとさらに効率的になります)

再帰呼び出しの基本については以上となりますが、ここで注意してほしいのは、関数型言語には(基本的には)ループ文というものがない、ということです。階乗計算ならループ文を使えば簡単なのに、と思うかもしれませんが、ループ文は基本的に変数代入による計算を想定しているため、代入操作をもたない(純粋な)関数型言語にはそぐわない、ということです。ループ一つ書くためにいちいち再帰呼び出しするのはかなり面倒だと感じるかもしれませんが、今後リストやリストに対する高階関数を勉強することでもう少しこの面倒さが軽減されるようになります。

※ 名前付き let というループ処理もありますが、却ってわかりにくくなってしまうためこの講義ではこれを使わないようにします。レポート課題の解答で名前付き let は絶対使わないようにしてください。

1.2.2 木構造再帰

計算でよくある再帰は木構造再帰 (tree recursion)です。フィボナッチ数列を例に木構造再帰について説明します。まずフィボナッチ数列は次のように定義が与えられます。

$$a_0 = 0$$
$$a_1 = 1$$
$$a_n = a_{n-1} + a_{n-2} \ (if\ n \geq 2)$$

この定義に従うとフィボナッチ数列$a_0, a_1, a_2, \cdots$は次のようになります。

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, \cdots$$

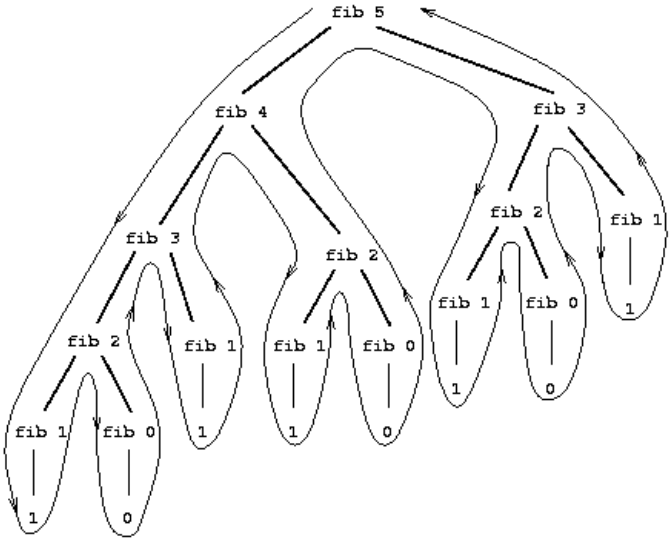
フィボナッチ数列の定義を関数として書き直すと次のようになります。

$$fib(n) = \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ fib(n-1) + fib(n-2) & if\ n \geq 2 \end{cases}$$

フィボナッチ数列をこのとおりに実装すると次のようなプログラムになります。

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

このプロセスは図で示すと次のようになっています。



しかし、この方法は明らかに重複した計算を行っていて効率が悪い。そこで、この木構造再帰的プロセスを線形反復で計算するようにすれば効率がかなり良くなります。

```
(define (fib n)
  (fib-iter 1 0 n))
(define (fib-iter a b count)
  (if (= count 0)
     b
     (fib-iter (+ a b) a (- count 1))))
```

1.2.4 べき乗

次にべき乗の計算について考えてみましょう。再帰的プロセスと反復的プロセスでそれぞれ定義してみよう。

・再帰的プロセス

```
(define (expt b n)
  (if (= n 0)
     1
     (* b (expt b (- n 1)))))
```

・反復的プロセス

```
(define (expt b n)
  (expt-iter b n 1))
(define (expt-iter b counter product)
  (if (= counter 0)
     product
     (expt-iter b (- counter 1) (* b product))))
```

しかし、この計算はもっと効率化できます。例えば、b^8は

$$b^2 = bb$$

$$b^4 = b^2 b^2$$

$$b^8 = b^4 b^4$$

と計算すれば3ステップで計算することができます。一般に、

$b^n = \left(b^{n/2}\right)^2$　　偶数の時

$b^n = bb^{n-1}$　　奇数の時

と計算すればうまくいきます。

```
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (square (fast-expt b (/ n 2))))
        (else (* b (fast-expt b (- n 1))))))
(define (even? n)
  (= (remainder n 2) 0))
```

とすれば、対数的なステップ数でべき乗を計算することができます。