# 原始帰納的関数

**原始帰納的関数（primitive recursive function）**の定義は、教科書によって若干異なるのですが、どれをもとにしても結果には影響はありません。この講義では、以下のような定義を採用します。

**定義**　自然数 $m, n$ について、**定数関数** $K_m^n : \mathbf{N}^n \to \mathbf{N}$ を

$$K_m^n(x_1, \ldots, x_n) = m$$

で定める。

**定義**　自然数 $1 \leq i \leq n$ について、**射影関数** $P_i^n : \mathbf{N}^n \to \mathbf{N}$ を

$$P_i^n(x_1, x_2, \ldots, x_n) = x_i$$

で定める。特に $P_1^1 : \mathbf{N} \to \mathbf{N}$ は恒等関数（入力をそのまま出力する関数）である。

**定義**　**後者関数 (successor)** $S : \mathbf{N} \to \mathbf{N}$ を

$$S(x) = x + 1$$

で定める。

**定義**　**原始帰納的関数**とは、以下の 1.〜3. から得られる自然数上の関数である。

1. **（初期関数）**定数関数 $K_m^n$, 射影関数 $P_i^n$, 後者関数 $S$ は原始帰納的関数である。
2. **（関数合成）**$f : \mathbf{N}^m \to \mathbf{N}$ と $f_i : \mathbf{N}^n \to \mathbf{N}$ $(i = 1, 2, \ldots, m)$ が原始帰納的関数であるとき

   $$g(x_1, \ldots, x_n) = f(f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n))$$

   で定義される関数 $g : \mathbf{N}^n \to \mathbf{N}$ は原始帰納的関数である。
3. **（原始帰納法）**$f : \mathbf{N}^n \to \mathbf{N}$ と $g : \mathbf{N}^{n+2} \to \mathbf{N}$ が原始帰納的関数であるとき

   $$\begin{aligned} h(x_1, \ldots, x_n, 0) &= f(x_1, \ldots, x_n) \\ h(x_1, \ldots, x_n, y+1) &= g(x_1, \ldots, x_n, y, h(x_1, \ldots, x_n, y)) \end{aligned}$$

   で定義される関数 $h : \mathbf{N}^{n+1} \to \mathbf{N}$ は原始帰納的関数である。

特に $n = 0$ のとき、$\mathbf{N}^0 \to \mathbf{N}$ を $\mathbf{N}$ と同一視することにより、3. は以下のようになる:

自然数 $m$ と原始帰納的関数 $g : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ について

$$\begin{aligned} h(0) &= m \\ h(y+1) &= g(y, h(y)) \end{aligned}$$

で定義される関数 $h : \mathbf{N} \to \mathbf{N}$ は原始帰納的関数である。

**原始帰納的関数の例**　原始帰納的関数は、すべて（直感的な意味で）「計算できる」。初期関数はおそらく誰にとっても計算可能であるし、計算可能な関数の合成も当然計算可能である。また、原始帰納法は、入力（のうちのひとつ）について帰納法により計算を定義するというものであり、計算するうえで特に難しいことはない。実際、原始帰納的関数を計算するプログラムを書くことは簡単である。また、原始帰納的関数の計算は、必ず停止する（原始帰納的関数の構成に関する帰納法で証明できる）。

**例：足し算**　$\mathsf{add} : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ $(\mathsf{add}(x, y) = x + y)$ は原始帰納的関数である。

$$\begin{array}{lcll} \mathsf{add}(x, 0) & = & P^1_1(x) & (= x) \\ \mathsf{add}(x, y+1) & = & S(P^3_3(x, y, \mathsf{add}(x, y))) & (= \mathsf{add}(x, y) + 1) \end{array}$$

詳しく説明すると、まず $P^1_1 : \mathbf{N} \to \mathbf{N}$ は 1. より原始帰納的関数である。また、$S : \mathbf{N} \to \mathbf{N}$ や、$P^3_3 : \mathbf{N} \times \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ も 1. より原始帰納的関数である。したがって、

$$g(x_1, x_2, x_3) = S(P^3_3(x_1, x_2, x_3))$$

で定義される、合成関数 $g : \mathbf{N} \times \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ も 2. より原始帰納的関数である。最後に、3. を用いて、$P^1_1 : \mathbf{N} \to \mathbf{N}$ と $g : \mathbf{N} \times \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ から

$$\begin{array}{lcl} \mathsf{add}(x, 0) & = & P^1_1(x) \\ \mathsf{add}(x, y+1) & = & g(x, y, \mathsf{add}(x, y)) \quad = \quad S(P^3_3(x, y, \mathsf{add}(x, y))) \end{array}$$

で定義される $\mathsf{add} : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ も原始帰納的関数であることがわかる。この add が $\mathsf{add}(x, y) = x + y$ をみたすことは、帰納法で容易に証明できる。
OCaml [1]というプログラミング言語で add を書くと

```
let rec add(x,y) = if y=0 then x else add(x,y-1)+1;;
```

**例：掛け算**　$\mathsf{mul} : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ $(\mathsf{mul}(x, y) = x \times y)$ は原始帰納的関数である。

$$\begin{array}{lcll} \mathsf{mul}(x, 0) & = & K^1_0(x) & (= 0) \\ \mathsf{mul}(x, y+1) & = & \mathsf{add}(P^3_1(x, y, \mathsf{mul}(x, y)), P^3_3(x, y, \mathsf{mul}(x, y))) & (= \mathsf{add}(x, \mathsf{mul}(x, y))) \end{array}$$

OCaml のプログラムで書くと

```
let rec mul(x,y) = if y=0 then 0 else add(x,mul(x,y-1));;
```

**例：前者関数 (predcessor)**　$\mathsf{pre} : \mathbf{N} \to \mathbf{N}$ $(\mathsf{pre}(0) = 0, \mathsf{pre}(x+1) = x)$ は原始帰納的関数である。

$$\begin{array}{lcll} \mathsf{pre}(0) & = & 0 \\ \mathsf{pre}(y+1) & = & P^2_1(y, \mathsf{pre}(y)) & (= y) \end{array}$$

$\mathsf{pre}(y+1)$ の計算のなかで、$\mathsf{pre}(y)$ は実際には用いられていないことに注意。

[1] OCaml については、例えば五十嵐淳「プログラミング in OCaml」（技術評論社 2007）を参照。

ところで、原始帰納的関数を与える際に、関数合成のためにいちいち各関数の変数の数を（射影関数を用いて）揃えるのは面倒である。実際には、以下のように、もっと自由なかたちで関数合成を行なってかまわない。

**問題**　(2. の一般化)

(i) $f : \mathbf{N}^m \to \mathbf{N}$ と $f_i : \mathbf{N}^{n_i} \to \mathbf{N}$ $(i = 1, 2, \ldots, m)$ が原始帰納的関数であるとき、

$$g(x_1, \ldots, x_k) = f(f_1(x_{a_{1,1}}, \ldots, x_{a_{1,n_1}}), \ldots, f_m(x_{a_{m,1}}, \ldots, x_{a_{m,n_m}}))$$

（ただし $1 \leq a_{i,j} \leq k$）で定義される関数 $g : \mathbf{N}^k \to \mathbf{N}$ は原始帰納的関数であることを示せ。

(ii)　((i) の特殊な場合）$f : \mathbf{N}^n \to \mathbf{N}$ が原始帰納的関数であるとき、

$$g(x_1, \ldots, x_m) = f(x_{b_1}, \ldots, x_{b_n})$$

（ただし $1 \leq b_i \leq m$）で定義される関数 $g : \mathbf{N}^m \to \mathbf{N}$ は原始帰納的関数であることを示せ。

**練習問題**

| | | |
|---|---|---|
| 引き算 | $\mathsf{sub} : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ | $\mathsf{sub}(x, y) = \begin{cases} x - y & (x \geq y) \\ 0 & (x < y) \end{cases}$ |
| べき | $\mathsf{exp} : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ | $\mathsf{exp}(x, y) = x^y$ |
| 階乗 | $\mathsf{fac} : \mathbf{N} \to \mathbf{N}$ | $\mathsf{fac}(x) = x! = x \times (x-1) \times \ldots \times 1$ |
| 最大値 | $\mathsf{max} : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ | $\mathsf{max}(x, y) = \begin{cases} x & (x \geq y) \\ y & (x < y) \end{cases}$ |
| 最小値 | $\mathsf{min} : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ | $\mathsf{min}(x, y) = \begin{cases} y & (x \geq y) \\ x & (x < y) \end{cases}$ |

はすべて原始帰納的関数であることを示せ。

**計算可能＝原始帰納的？**　さて、原始帰納的関数は計算可能である（プログラムできる）が、逆に、計算可能な自然数上の関数はどれも原始帰納的関数だろうか?

答え：**NO！** 原始帰納的関数でないが計算可能な関数が存在する。

**例：アッカーマン関数 (Ackermann function)**
以下のように、関数 $\mathsf{ack} : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$ を定義する。

$$\begin{array}{lcl} \mathsf{ack}(0, n) & = & n + 1 \\ \mathsf{ack}(m + 1, 0) & = & \mathsf{ack}(m, 1) \\ \mathsf{ack}(m + 1, n + 1) & = & \mathsf{ack}(m, \mathsf{ack}(m + 1, n)) \end{array}$$

ack は計算可能であるが、原始帰納的関数ではない。

**問題（易）** ack を計算するプログラムを書け。また、実際に実行してみよ。

```
/* Sample Code in Java. Example: ‘‘java Ack 2 4’’ returns 11 */

 public class Ack {
   public static long ack(long m, long n) {
      if (m == 0) return n + 1;
      if (n == 0) return ack(m - 1, 1);
      return ack(m - 1, ack(m, n - 1));
   }
   public static void main(String[] args) {
      long M = Long.parseLong(args[0]);
      long N = Long.parseLong(args[1]);
      System.out.println(ack(M, N));
   }
 }


(* Sample Code in OCaml. Example: ‘‘ack 2 4;;’’ returns 11 *)

 let rec ack m n =
     if m = 0 then n + 1
              else if n = 0 then ack (m - 1) 1
                            else ack (m - 1) (ack m (n - 1));;
```

**問題（易）** $\mathsf{ack}(1, y) = y + 2$, $\mathsf{ack}(2, y) = 2 \times y + 3$ であることを示せ。

**問題（並）** どの $x, y \in \mathbf{N}$ についても $\mathsf{ack}(x, y)$ の値は必ず有限回の計算で求められることを示せ。

**問題（難）** ack が原始帰納的関数で**ない**ことを示せ。
ヒント：準備として、以下の１ー４を順に示し、その結果を用いる。

1. $x + y < \mathsf{ack}(x, y)$
2. $\mathsf{ack}(x, y) < \mathsf{ack}(x, y + 1) \leq \mathsf{ack}(x + 1, y)$
3. 任意の $a, b \in \mathbf{N}$ について $\mathsf{ack}(a, \mathsf{ack}(b, y)) < \mathsf{ack}(c, y)$ を満たすような $c \in \mathbf{N}$ が存在する
4. $f : \mathbf{N}^n \to \mathbf{N}$ が原始帰納的関数ならば $f(x_1, \ldots, x_n) < \mathsf{ack}(c, x_1 + \ldots + x_n)$ を満たすような $c \in \mathbf{N}$ が存在する。

どうやら、原始帰納的関数は、「計算可能である」という性質を正確につかまえてはなさそうである。何かが足りない...