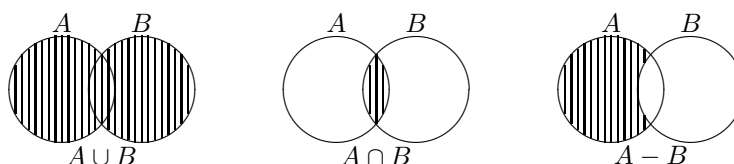


数学的な準備

集合 集合とは、ものの集まりである。この講義では、以下のような一般的な記法を用いる。

- \mathbf{N} — 自然数 $0, 1, 2, 3, \dots$ 全体の集合 (0 も含む)
- $\{a, b, c\}$ — a, b, c を要素にもつ集合
- \emptyset — 空集合
- $x \in A$ — x は集合 A の要素
- $\{x \in A \mid P(x)\}$ — 性質 $P(x)$ を満たすような集合 A の要素 x を集めてできる集合 (たとえば、 $\{x \in \mathbf{R} \mid x^2 + x - 2 = 0\}$ は、 $x^2 + x - 2 = 0$ をみたす実数 x 全体の集合)
- $A \subseteq B$ — A は B の部分集合 ($x \in A$ ならば $x \in B$)
- $A \cup B$ — 集合 A と B の和集合
- $A \cap B$ — 集合 A と B の共通部分
- $A - B$ — 集合 A の要素のうち、集合 B に含まれるものを除いた集合、すなわち $\{a \in A \mid a \notin B\}$



- $A \times B$ — 集合 A と B の直積集合 (A の要素と B の要素の組を集めてできる集合)。 $A \times B$ の要素は (a, b) ($a \in A, b \in B$) と表せる。
- A^n — 集合 A の n 個の直積集合 ($\overbrace{A \times A \times \dots \times A}^n$) (A の要素を n 個並べたリストを集めてできる集合)。例: $A^1 = A, A^2 = A \times A, A^3 = A \times A \times A$ 。
 A^n の要素は (a_1, \dots, a_n) ($a_i \in A$) と表せる。特に、 A^0 は空のリスト $()$ だけを要素にもつ一点集合である。この定義により、 A^{m+n} と $A^m \times A^n$ を同一視することができる。
- 集合 A から集合 B への関数全体のなす集合を $A \rightarrow B$ であらわす。 f が集合 A から集合 B への関数であることを $f: A \rightarrow B$ と書く。関数 $f: A \rightarrow B$ については、任意の $a \in A$ に対し、 $f(a) = b$ となるような $b \in B$ がただひとつ存在する。(より正確には、集合 A から集合 B への関数 f は、任意の $a \in A$ に対し $(a, b) \in f$ となる $b \in B$ がただひとつ存在するような $A \times B$ の部分集合であると定義される。この定義から、空集合 \emptyset から集合 B への関数はひとつだけ存在することがわかる。)
- 集合 A の部分集合全体からなる集合を A の巾集合と呼び、 2^A であらわす。たとえば、 $2^{\{a, b\}} = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$ 、また $2^\emptyset = \{\emptyset\}$ である。

可算集合 可算集合とは、空集合であるか、またはその要素を

$$a_0, a_1, a_2, a_3, \dots$$

というように、自然数の添字をつけてすべて並べあげる（列挙する）ことができる集合のことをいう。（すなわち、可算集合とは、空集合か、あるいは \mathbf{N} からの全射的な関数が存在するような集合のことである。）

- 有限集合（要素が有限個しかない集合）は可算集合である。
- \mathbf{N} は可算集合である。
- 整数全体の集合 \mathbf{Z} や有理数全体の集合 \mathbf{Q} は可算集合である。
- $\mathbf{N} \times \mathbf{N}$ は可算集合である。
- 実数全体の集合 \mathbf{R} は可算集合ではない。
- $[0, 1] = \{x \in \mathbf{R} \mid 0 \leq x \leq 1\}$ は可算集合ではない。
- \mathbf{N} から \mathbf{N} への関数全体の集合 $\mathbf{N} \rightarrow \mathbf{N}$ は可算集合ではない。
- \mathbf{N} の部分集合全体の集合 $2^{\mathbf{N}}$ は可算集合ではない。

可算集合とは、有限集合か、または自然数で数え上げられるような（小さい）無限集合であると理解できる。可算でない集合は、可算集合より真に大きい無限集合である。（正確には、この「大きさ」は、集合の濃度または基数と呼ばれる概念についてのものである。）

対角線論法 $\mathbf{N} \rightarrow \mathbf{N}$ が可算集合ではないことの、カントールの対角線論法による証明は以下のとおり。

$\mathbf{N} \rightarrow \mathbf{N}$ が可算集合だと仮定すると、その要素を

$$f_0, f_1, f_2, f_3, \dots$$

というように、すべて並べあげることができる。ここで、 $g: \mathbf{N} \rightarrow \mathbf{N}$ を

$$g(x) = f_x(x) + 1$$

で定義する。すると、 g はどの f_n とも異なる。実際、任意の自然数 n について、 $g(n) = f_n(n) + 1 \neq f_n(n)$ なので g と f_n は異なる関数である（下図を参照）。

	0	1	2	3	...
f_0	$f_0(0)$	$f_0(1)$	$f_0(2)$	$f_0(3)$...
f_1	$f_1(0)$	$f_1(1)$	$f_1(2)$	$f_1(3)$...
f_2	$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_2(3)$...
f_3	$f_3(0)$	$f_3(1)$	$f_3(2)$	$f_3(3)$...
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots
g	$f_0(0)+1$	$f_1(1)+1$	$f_2(2)+1$	$f_3(3)+1$...

したがって、 $\mathbf{N} \rightarrow \mathbf{N}$ の要素を自然数の添字をつけてすべて数え上げることはできない。すなわち、 $\mathbf{N} \rightarrow \mathbf{N}$ は可算集合ではない。

対角線論法に関する話題を解説した資料「自己言及の論理と計算」が長谷川のページ

<http://www.kurims.kyoto-u.ac.jp/~hassei/index-j.html>

から入手可能です。

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 $K_m^n : \mathbf{N}^n \rightarrow \mathbf{N}$ を

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

で定める。

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

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

で定める。特に $P_1^1 : \mathbf{N} \rightarrow \mathbf{N}$ 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) $S : \mathbf{N} \rightarrow \mathbf{N}$ 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。

- (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。
- (関数合成) $f : \mathbf{N}^m \rightarrow \mathbf{N}$ と $f_i : \mathbf{N}^n \rightarrow \mathbf{N}$ ($i = 1, 2, \dots, m$) が原始帰納的関数であるとき

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

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

- (原始帰納法) $f : \mathbf{N}^n \rightarrow \mathbf{N}$ と $g : \mathbf{N}^{n+2} \rightarrow \mathbf{N}$ が原始帰納的関数であるとき

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

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

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

自然数 m と原始帰納的関数 $g : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ 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 $h : \mathbf{N} \rightarrow \mathbf{N}$ 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 $\text{add} : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ ($\text{add}(x, y) = x + y$) は原始帰納的関数である。

$$\begin{aligned} \text{add}(x, 0) &= P_1^1(x) && (= x) \\ \text{add}(x, y + 1) &= S(P_3^3(x, y, \text{add}(x, y))) && (= \text{add}(x, y) + 1) \end{aligned}$$

詳しく説明すると、まず $P_1^1 : \mathbf{N} \rightarrow \mathbf{N}$ は 1. より原始帰納的関数である。また、 $S : \mathbf{N} \rightarrow \mathbf{N}$ や、 $P_3^3 : \mathbf{N} \times \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ 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 $g : \mathbf{N} \times \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ も 2. より原始帰納的関数である。最後に、3. を用いて、 $P_1^1 : \mathbf{N} \rightarrow \mathbf{N}$ と $g : \mathbf{N} \times \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ から

$$\begin{aligned} \text{add}(x, 0) &= P_1^1(x) \\ \text{add}(x, y + 1) &= g(x, y, \text{add}(x, y)) = S(P_3^3(x, y, \text{add}(x, y))) \end{aligned}$$

で定義される $\text{add} : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ も原始帰納的関数であることがわかる。この add が $\text{add}(x, y) = x + y$ をみたすことは、帰納法で容易に証明できる。

OCaml¹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 $\text{mul} : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ ($\text{mul}(x, y) = x \times y$) は原始帰納的関数である。

$$\begin{aligned} \text{mul}(x, 0) &= K_0^1(x) && (= 0) \\ \text{mul}(x, y + 1) &= \text{add}(P_1^3(x, y, \text{mul}(x, y)), P_3^3(x, y, \text{mul}(x, y))) && (= \text{add}(x, \text{mul}(x, y))) \end{aligned}$$

(ここで、 $g(x_1, x_2, x_3) = \text{add}(P_1^3(x_1, x_2, x_3), P_3^3(x_1, x_2, x_3))$ とすれば、二行目は $\text{mul}(x, y + 1) = g(x, y, \text{mul}(x, y))$ となることに注意。)

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 (predecessor) $\text{pre} : \mathbf{N} \rightarrow \mathbf{N}$ ($\text{pre}(0) = 0, \text{pre}(x + 1) = x$) は原始帰納的関数である。

$$\begin{aligned} \text{pre}(0) &= 0 \\ \text{pre}(y + 1) &= P_1^2(y, \text{pre}(y)) && (= y) \end{aligned}$$

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

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もかまわない。

問題 (2. の一般化)

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

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

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

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

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

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

練習問題

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

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 $\text{ack} : \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$ を定義する。

$$\begin{aligned} \text{ack}(0, n) &= n + 1 \\ \text{ack}(m + 1, 0) &= \text{ack}(m, 1) \\ \text{ack}(m + 1, n + 1) &= \text{ack}(m, \text{ack}(m + 1, n)) \end{aligned}$$

ack は計算可能であるが、原始帰納的関数ではない。

¹OCaml 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。

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

問題（並） どの $x, y \in \mathbf{N}$ についても $\text{ack}(x, y)$ 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1-4を順に示し、その結果を用いる。

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

帰納的関数

帰納的関数 (recursive function) の定義も、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 (ただし後の注意も読んでください)。この講義では、以下のような定義を採用します。

定義 集合 A から集合 B への部分関数とは、定義域が A の部分集合、値域が B であるような関数である。 f が A から B への部分関数であることを、 $f: A \rightarrow B$ とあらわす。 $a \in A$ が $f: A \rightarrow B$ の定義域に含まれていないとき、 $f(a)$ は未定義であるという。

(特に、 A から B への関数は、定義域が A 全体であるような A から B への部分関数である。)

定義 部分関数 $f: \mathbf{N}^{n+1} \rightarrow \mathbf{N}$ について、部分関数 $\mu(f): \mathbf{N}^n \rightarrow \mathbf{N}$ を以下のように定める。 $\mu(f)$ の定義域は

$$\{(x_1, \dots, x_n) \in \mathbf{N}^n \mid \exists y \in \mathbf{N} f(x_1, \dots, x_n, y) = 0 \ \& \ \forall z < y f(x_1, \dots, x_n, z) > 0\}$$

である。 (x_1, \dots, x_n) が $\mu(f)$ の定義域に属しているとき、

$$\mu(f)(x_1, \dots, x_n) = f(x_1, \dots, x_n, y) = 0 \text{ を満たすような最小の自然数 } y$$

とする。 (x_1, \dots, x_n) が $\mu(f)$ の定義域に属していないときには、 $\mu(f)(x_1, \dots, x_n)$ は未定義である。

例 $f(5, 0) = 2, f(5, 1) = 1, f(5, 2) = 0, \dots$ なら、 $\mu(f)(5) = 2$ である。しかし、 $g(5, 0) = 2, g(5, 1)$ は未定義、 $g(5, 2) = 0, \dots$ なら、 $\mu(g)(5)$ は未定義である。

参考 OCaml では ($n = 3$ の場合だと)

```
let mu(f)(x1,x2,x3) =
  let y = ref 0 in
  while (f(x1,x2,x3,!y) > 0) do
    y := !y + 1;
  done;
  !y;
```

$\mu(f)$ を、 f の最小解関数 (minimization) と呼ぶ。 $\mu(f)(x_1, \dots, x_n)$ のかわりに、 $\mu y. f(x_1, \dots, x_n, y)$ と書くこともある。(なお、論理学では、ギリシャ文字 μ を、「... を満たす最小のもの」をあらわすのに使うことが多い。)

定義 帰納的関数とは、以下の 1.~4. から得られる自然数上の部分関数である。

1. 初期関数 (原始帰納的関数の定義の 1. を参照) は帰納的関数である。
2. 帰納的関数から関数合成 (原始帰納的関数の定義の 2. を参照) を用いて得られる部分関数は帰納的関数である。
3. 帰納的関数から原始帰納法 (原始帰納的関数の定義の 3. を参照) を用いて得られる部分関数は帰納的関数である。
4. $f: \mathbf{N}^{n+1} \rightarrow \mathbf{N}$ が帰納的関数であるとき、 f の最小解関数 $\mu(f): \mathbf{N}^n \rightarrow \mathbf{N}$ は帰納的関数である。

なお、4. は「 $f: \mathbf{N}^{n+1} \rightarrow \mathbf{N}$ が原始帰納的関数であるとき、 $\mu(f): \mathbf{N}^n \rightarrow \mathbf{N}$ は帰納的関数である。」と制限しても実はかまわない。

注意 ここで定義された帰納的関数は、部分関数であることを強調して、**部分帰納的関数 (partial recursive function)** とも呼ばれる。帰納的関数 $f: \mathbf{N}^n \rightarrow \mathbf{N}$ が関数であるとき、すなわき f の定義域が \mathbf{N}^n 全体であるとき、 f を**全域帰納的関数 (total recursive function)** と呼ぶ。教科書によっては、全域帰納的関数のことを帰納的関数と呼ぶ場合があるので、注意が必要である。

チャーチとチューリングの提唱

「アルゴリズム＝機械的に実行できる手続き」を数学的に正確に定式化する試みが、1920～30年代にいろいろなされてきた：

ゲーデル/クリネ	「帰納的関数」	(1934/1936)
チューリング	「チューリング機械」	(1936)
チャーチ	「ラムダ計算」	(1932)

ところが、実はこれらのすべてが同値であることがわかった。また、これらで表現できない計算可能な関数は見つからなかった。これらの経験的事実から、チャーチ（と、独立にチューリング）は、

「計算可能な関数とは帰納的関数である」

と主張した（1936）。これを「チャーチ・チューリングの提唱」（Church-Turing Thesis）あるいは「チャーチの提唱」「チューリングの提唱」と呼ぶ。

決定可能性

「計算可能な（部分）関数」として帰納的関数を考えるのに対応して、「計算可能な（あるいは決定可能な）集合」として帰納的集合という概念を導入する。

定義 \mathbf{N}^n の部分集合 A について、関数 $\chi_A : \mathbf{N}^n \rightarrow \mathbf{N}$ を

$$\chi_A(x_1, \dots, x_n) = \begin{cases} 0 & (x_1, \dots, x_n) \in A \\ 1 & (x_1, \dots, x_n) \notin A \end{cases}$$

で定義する。 χ_A を A の**特性関数**（characteristic function）という。

定義

- \mathbf{N}^n の部分集合 A の特性関数が全域帰納的関数であるとき、 A は**帰納的集合**（recursive set）であるという。
- \mathbf{N}^n の部分集合 A がある（部分）帰納的関数の定義域になっているとき、 A は**帰納的可算集合**（recursively enumerable set）であるという。

チャーチの提唱を受け入れるならば、帰納的集合とは、要素であるかないかを機械的に判定することが可能（**決定可能**、decidable）な集合である。また、帰納的可算集合とは、要素である場合には停止するが、要素でない場合には停止しない（したがって何もわからない）ような機械的な手続きがあるような集合である（**半決定可能**、semidecidable という）。

補題 有限集合は帰納的集合である。（したがって、帰納的集合でないような集合は、必ず無限集合である。）

補題 \mathbf{N}^n の部分集合 A について、 A が帰納的集合であるならば、 A は帰納的可算集合でもある。

補題 \mathbf{N}^n の部分集合 A について、 A が帰納的集合であるならば、 $\mathbf{N}^n - A$ も帰納的集合である。

定理 \mathbf{N}^n の部分集合 A について、 A が帰納的集合であることと、 A と $\mathbf{N}^n - A$ の両方が帰納的可算集合であることは同値である。

算術化、停止性問題

帰納的関数は、初期関数から関数合成と最小解の有限の組み合わせで得られるので、その構成方法に適当に自然数を対応させて数え上げることができる（**算術化**または**ゲーデル化**と呼ばれる）。（したがって、帰納的関数全体は可算集合である。）以下の定理は、その数え上げを遂行する帰納的関数が存在する（したがって計算可能である）ことを主張している。

定理（数え上げ） $n \geq 1$ について、以下のような性質を満たす帰納的関数 $\Phi^n : \mathbf{N}^{n+1} \rightarrow \mathbf{N}$ が存在する。

任意の帰納的関数 $f : \mathbf{N}^n \rightarrow \mathbf{N}$ について、ある自然数 i が存在して $f(x_1, \dots, x_n) = \Phi^n(i, x_1, \dots, x_n)$ が成り立つ。

$f(x_1, \dots, x_n) = \Phi^n(i, x_1, \dots, x_n)$ であるような i を、 f のインデックスとよぶ。また、 Φ^n は、(n 変数の帰納的関数に関する) **万能関数 (universal function)** とよばれる。

なお、インデックスは、ただひとつに決まるわけではない。実は、どの帰納的関数も無限個のインデックスを持つ。(言い換えると、どの計算可能性関数にも、対応するプログラムが無限に存在する。)

定理（停止性問題） $\mathbf{N} \times \mathbf{N}$ の部分集合 $\{(i, x) \in \mathbf{N}^2 \mid \Phi^1(i, x) \text{ が定義されている}\}$ は帰納的集合ではない (すなわち決定不可能である)。

すなわち、与えられたプログラム i と入力値 x から、その実行 $\Phi^1(i, x)$ が停止するかどうか判定するプログラムは存在しない。

参考文献 日本語の教科書では、以下のものがよいと思います。
広瀬 健, 帰納的関数. 共立出版, 1989.
高橋 正子, 計算論 計算可能性とラムダ計算. 近代科学社, 1991.