第4章　帰納的定義と帰納法

この章では，形式的な定義方法・推論方法としての「帰納」 (induction) について学ぶ．

よく知られた数学的帰納法 (mathematical induction) は,「全ての自然数に対して，性質 P が成立する」ことを示すための推論法である．これは，まず「P(0)」 が成立することを示す，次に,「P(x) を仮定して，P(x+1) を示す」ことにより,「全ての自然数 n に対して，P(n) が成立する」ことが言える，という推論である．

しかし，帰納法は，自然数に対する推論法だけではない．自然数以外にも，リストや木などの集合を「帰納的に定義」することができ，それぞれに対する推論法としての帰納法がある．

実は，数学的に厳密な意味で，無限に多くの要素を持つ集合を作りだしたり，無限に多くの要素に対する性質を示す方法というのは，それほどたくさんあるわけではない．むしろ，そのような定義・推論方法は，ほとんどの場合,「帰納」によると言える．

4.1　帰納的に定義された集合

「自然数の集合」などの無限集合を厳密に定義するために，帰納的定義 (inductive definition) を用いる．

集合 A の帰納的定義とは，以下のように集合 A を定義する方法である．

- (basis，基礎) いくつかのもの (あらかじめわかっているもの) は，集合 A の要素であることを定める．
- (induction step，ステップ) すでに A の要素であることがわかっているものから，新たな A の要素を作る操作を定める．
- (closure，限定句) 上の操作を，有限回適用して作られた要素のみが A の要素であると定める．

なお，帰納的定義では，closure 条件を常に必要とするので，省略して書かないことも多い．

例 81 3 以上の奇数の集合 A の帰納的定義．

- $3 \in A$
- $x \in A \Rightarrow x + 2 \in A$
- 上記の操作で作られるものだけが A の要素である．(あるいは，A は上記を満たす最小の集合である．)

なお，3 番目の closure 条件がないと，集合 A が一意に定まらない．たとえば，自然数の集合 $\mathcal{N}$ も 1, 2 番目の条件を満たしている.「定義」であるためには，一意に定まらなければ意味がないため，帰納的定義においては，closure 条件の記述を省略してあっても必ず設定していると考える．

例 82 自然数の集合 $\mathcal{N}$.

- $0 \in \mathcal{N}$.
- $n \in \mathcal{N} \Rightarrow n+1 \in \mathcal{N}$

例 83 $A = \{1, 3, 7, 15, 31, \ldots\}$

- $1 \in A$.
- $x \in A \Rightarrow 2x+1 \in A$

4.1.1 リスト

リスト (list) は，列 (sequence) とも呼ばれ，コンピュータ科学で頻繁に現れるデータ構造である．

リストの非形式的な (厳密でない) 定義: 集合 A の要素からなる任意の長さの組を A のリストという．すなわち，A のリストの集合 $List_A$ は以下のように定義される．

$$List_A = \{\langle x_1, \ldots, x_n\rangle \mid n \geq 0 \wedge \forall i.(1 \leq i \leq n \Rightarrow x_i \in A)\}$$

ただし，この定義は『...』を使っているので，厳密な定義ではない．

例 84 自然数のリスト:

$$\langle\rangle, \langle 1, 2\rangle, \langle 2\rangle, \langle 3, 2, 1\rangle$$

長さが 0 のリスト $\langle\rangle$ を空リストと呼ぶ．これを nil と書くことがある．

リストに対する基本的な操作として cons, head, tail がある．

$$\mathrm{cons}(x, \langle x_1, \ldots, x_n\rangle) = \langle x, x_1, \ldots, x_n\rangle$$

$$\mathrm{head}(\langle x_1, \ldots, x_n\rangle) = x_1$$

$$\mathrm{tail}(\langle x_1, \ldots, x_n\rangle) = \langle x_2, \ldots, x_n\rangle$$

head と tail は空リストに対しては定義できないので，これらは ($List_A$ を定義域と考えた場合) 関数ではなく部分関数である．

例 85

$$\mathrm{head}(\langle a, b, c\rangle) = a$$

$$\mathrm{tail}(\langle a, b, c\rangle) = \langle b, c\rangle$$

$$\mathrm{cons}(a, \langle\rangle) = \langle a\rangle$$

$$\mathrm{cons}(a, \langle b, c\rangle) = \langle a, b, c\rangle$$

任意の $x \in A$ と $L \in List_A$ に対して，$x = \mathrm{head}(\mathrm{cons}(x, L))$ と $L = \mathrm{tail}(\mathrm{cons}(x, L))$ が成り立つ．また，L が空でないリストのとき，$L = \mathrm{cons}(\mathrm{head}(L), \mathrm{tail}(L))$ が成り立つ．

上記の定義では，リストを組と考え，cons 等はそれに対する操作であったが，逆に，cons 操作を基本操作と考えることにより，リストの集合を帰納的に定義することができる．

リストの厳密な定義: 集合 A に対して，A のリストの集合 $List_A$ は，以下のように帰納的に定義される．

- $\langle\rangle \in List_A$

- $x \in A \wedge L \in List_A \Rightarrow \mathrm{cons}(x, L) \in List_A$

$\mathrm{cons}(x, L)$ に対する省略記法として，中置演算子 :: を使って $x :: L$ と書くことにする．

$$\begin{aligned} & a :: (b :: (c :: \langle\rangle)) \\ = \ & \mathrm{cons}(a, \mathrm{cons}(b, \mathrm{cons}(c, \langle\rangle))) \\ = \ & \mathrm{cons}(a, \mathrm{cons}(b, \langle c \rangle)) \\ = \ & \mathrm{cons}(a, \langle b, c \rangle) \\ = \ & \langle a, b, c \rangle \end{aligned}$$

例 86 0 と 1 が交互に現れるリストの集合 S の帰納的定義

- $\langle 0 \rangle \in S$.
- $\langle 1 \rangle \in S$.
- $L \in S \wedge \mathrm{head}(L) = 0 \Rightarrow \mathrm{cons}(1, L) \in S$.
- $L \in S \wedge \mathrm{head}(L) = 1 \Rightarrow \mathrm{cons}(0, L) \in S$

この定義により，$S = \{\langle 0 \rangle, \langle 1 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 0, 1, 0 \rangle, \langle 1, 0, 1 \rangle, \ldots\}$ となる．

本講義では，リストを組と考え $\langle 1, 2, 3 \rangle$ のように表現するが，プログラミング言語によって異なる表記が用いられる．

- Lisp，Scheme 言語におけるリスト: (1 2 3)
- ML 言語におけるリスト: $[1, 2, 3]$

4.1.2 文字列

文字列 (string) もコンピュータ科学において重要なデータ構造である．この講義では文字列を abc のように書く [1]．長さが 0 の文字列，すなわち，空文字列を Λ で表す．

文字 (アルファベット) の集合を Σ と書くと，Σ 上の文字列とは，Σ に含まれる文字の (有限長の) 列のことである．

例 87 アルファベット $\{a, b\}$ 上の文字列.

$$\Lambda, a, b, aa, ab, ba, bb, \cdots$$

Σ 上の文字列の集合 Σ^* は，以下のように帰納的に定義できる．
(Σ 上の文字列の集合に対する定義 1)

- $\Lambda \in \Sigma^*$
- $x \in \Sigma \wedge s \in \Sigma^* \Rightarrow xs \in \Sigma^*$

文字列に対して，別の定義を考えることもできる．
(Σ 上の文字列の集合に対する定義 2)

[1] プログラミング言語では，文字列を表現する際は "abc" のように二重引用符で囲うことが多い．

- $\Lambda \in \Sigma^*$
- $x \in \Sigma \wedge s \in \Sigma^* \Rightarrow sx \in \Sigma^*$

(Σ 上の文字列の集合に対する定義 3)

- $\Lambda \in \Sigma^*$
- $x \in \Sigma \Rightarrow x \in \Sigma^*$
- $s \in \Sigma^* \wedge t \in \Sigma^* \Rightarrow st \in \Sigma^*$

これら 3 種類の定義は同等であることが証明できる.

定義 1・定義 2 においては，データが与えられたときに，それが Σ^* 上の文字列であることを導く方法は一意的である．たとえば， $bab \in \{a,b\}^*$ は次のように導ける.

- $\Lambda \in \{a,b\}^*$.
- これと $b \in \{a,b\}$ より，$b \in \{a,b\}^*$.
- これと $a \in \{a,b\}$ より，$ab \in \{a,b\}^*$.
- これと $b \in \{a,b\}$ より，$bab \in \{a,b\}^*$.

これ以外に $bab \in \{a,b\}^*$ を導く方法はない.

一方，定義 3 においては，$s,t,u \in \Sigma^*$ という 3 つの文字列に対して，それらを結合した stu が文字列であることを 2 通り (以上) の方法で導くことができる.

- 「$s \in \Sigma^*$」と「$t \in \Sigma^*$」から「$st \in \Sigma^*$」を導き，これと「$u \in \Sigma^*$」から,「$stu \in \Sigma^*$」を導く.
- 「$t \in \Sigma^*$」と「$u \in \Sigma^*$」から,「$tu \in \Sigma^*$」を導き，これと「$s \in \Sigma^*$」から,「$stu \in \Sigma^*$」を導く.

このように同一のデータに対して複数の導出がある事を「曖昧な導出を持つ」と言う．曖昧な導出がある場合は，$(ab)c$ や $a(bc)$ のように括弧をつけて区別する必要がある [2].

例 88 $A = \{0,1\}$ 上の文字列で，右端の文字のみが 0 である文字列の集合 L の帰納的定義.

- $0 \in L$.
- $s \in L \Rightarrow 1s \in L$.

例 89 $S = \{a, b, ab, ba, aab, bba, aaab, bbba, \ldots\}$ の帰納的定義.

- $a \in S$, $b \in S$.
- $$s \in S \Rightarrow \begin{cases} bs \in S & (s = a \text{ の時}) \\ as \in S & (s = b \text{ の時}) \\ as \in S & (s \neq a \text{ かつ } strhead(s) = a \text{ の時}) \\ bs \in S & (s \neq b \text{ かつ } strhead(s) = b \text{ の時}) \end{cases}$$

ここで $strhead$ は文字列の先頭の文字を取る操作 (部分関数) とする.

[2] 括弧をつけない表記を用いる場合は，どちらの省略形であるかを決める必要がある．論理式の場合，たとえば，$A \wedge B \wedge C$ は $(A \wedge B) \wedge C$ の省略である，と定めるのと同様である.

4.1.3 2 分木 (binary Tree)

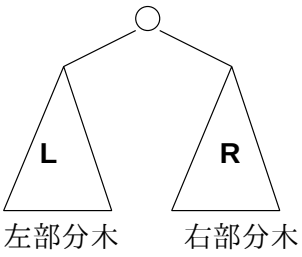
以前の章で，2 分木をグラフの一種として定義した．本節では，順序木としての 2 分木 (兄弟の間の順序を考慮に入れた 2 分木) の集合を帰納的に定義する．本節では，順序木しか扱わないので，単に 2 分木といえば順序木としての 2 分木である．

まず，単純な 2 分木，すなわち，2 分木の頂点にラベルがついていない 2 分木を定義する．葉を $\circ$ と表す．

定義 3 [単純な 2 分木 BinTree]

- $\circ \in \mathrm{BinTree}$
- $(L \in \mathrm{BinTree} \wedge R \in \mathrm{BinTree}) \Rightarrow \langle L, R\rangle \in \mathrm{BinTree}$.
- $L \in \mathrm{BinTree} \Rightarrow \langle L\rangle \in \mathrm{BinTree}$.

  最後の項は，子を 1 つだけ持つ節に対応する．(2 分木では，それぞれの節の子が 0〜2 個であるので，子が 1 個だけの節の場合も考慮する必要がある．)



ここで $\langle L, R\rangle$ は，2 分木 L と R を組み合わせて作った 2 分木である．この定義では，兄弟の順序関係を区別するため，$\langle L, R\rangle$ というように対を使っている．もし，兄弟の順序関係を区別しないのであれば $\{L, R\}$ と集合を使えばよい．木 $\langle L, R\rangle$ において，L のことを左部分木，R のことを右部分木という．

次に，ラベルつき 2 分木を定義する．すなわち，ラベル (名前) の集合を A とする時，2 分木の節 (葉でない頂点) に集合 A の要素が 1 つ付けられている 2 分木を定義する．

- $\circ \in \mathrm{BinTree}_A$
- $(x \in A \wedge L \in \mathrm{BinTree}_A \wedge R \in \mathrm{BinTree}_A) \Rightarrow \langle L, x, R\rangle \in \mathrm{BinTree}_A$.

  ここで x がこの節のラベル，L が左の部分木，R が右の部分木を表している．
- $(x \in A \wedge L \in \mathrm{BinTree}_A) \Rightarrow \langle L, x\rangle \in \mathrm{BinTree}_A$.

  これは，子が 1 つしかない節に対応している．

例 90 $\mathrm{BinTree}_{\mathcal{N}}$ の要素 (自然数のラベルがつけられた 2 分木)

$$\circ$$

$$\langle \circ, 1, \circ\rangle$$

$$\langle\langle \circ, 2, \circ\rangle, 1, \circ\rangle$$

$$\langle\langle \circ, 2\rangle, 1, \circ\rangle$$

以下では，ラベルつき 2 分木を表すとき，$\langle L, x, R\rangle$ の代わりに $\mathrm{Tree}(L, x, R)$ と書き，$\langle L, x\rangle$ の代わりに $\mathrm{Tree}(L, x)$ と書く．

例 91

$$\mathrm{Tree}(\mathrm{Tree}(\circ, 2, \circ), 1, \circ)$$

$$\mathrm{Tree}(\mathrm{Tree}(\circ, 2), 1, \circ)$$

リストに対する head, tail と同様に，ラベルつき 2 分木に対する操作を以下のように定義する．

$$\mathrm{label}(\mathrm{Tree}(t_1, x, t_2)) = x$$

$$\mathrm{label}(\mathrm{Tree}(t_1, x)) = x$$

$$\mathrm{left}(\mathrm{Tree}(t_1, x, t_2)) = t_1$$

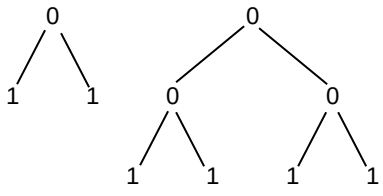
$$\mathrm{left}(\mathrm{Tree}(t_1, x)) = t_1$$

$$\mathrm{right}(\mathrm{Tree}(t_1, x, t_2)) = t_2$$

これらの操作は根だけからなる木に対しては定義されない．また right 操作は，根の子が 1 つしかない木に対しては定義されない．

例 92 $\mathrm{BinTree}_A$ のうち左右の部分木が同じ木の集合 T_A は以下のように定義できる．

- $\circ \in T_A$.
- $x \in A \wedge t \in T_A \Rightarrow \mathrm{Tree}(t, x, t) \in T_A$.

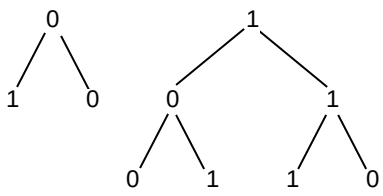


(葉を省略した．)

例 93 $\{0, 1\}$ の要素をラベルとして持つラベルつき 2 分木で，左右の部分木が同じ形をしているが，ラベルの $0, 1$ が入れ替わっている木の集合 $Opps$.

- $\circ, \langle\circ, 0, \circ\rangle, \langle\circ, 1, \circ\rangle \in Opps$.
-

$$x \in \{0,1\} \ \wedge \ t \in Opps \Rightarrow \begin{cases} \mathrm{Tree}(t, x, \mathrm{Tree}(\mathrm{right}(t), 1, \mathrm{left}(t))) \in Opps, & \text{if } \mathrm{label}(t) = 0 \\ \mathrm{Tree}(t, x, \mathrm{Tree}(\mathrm{right}(t), 0, \mathrm{left}(t))) \in Opps, & \text{if } \mathrm{label}(t) = 1 \end{cases}$$



(葉を省略した．)

4.1.4 BNF 記法 (†)

BNF 記法 (Backus-Naur Form または Backus Normal Form) は，言語の構文を簡潔に定義する方法の 1 つであり，特に，プログラミング言語の構文を記述する際によく用いられる．形式言語理論の言葉では，BNF 記法は文脈自由文法という種類の言語を定義するものであるが，これは帰納的定義の一種と見なすことができる．

ここでは，一般的な BNF 記法を定義するかわりに，簡単なプログラミング言語の構文の定義の例を与える．

例 94 [簡単なプログラム言語の構文]

変数の集合と定数の集合はあらかじめ定まっているものとする．たとえば，$\{a, b, c, \cdots, z\}$ 上の文字列を変数とし，$\{0, 1, 2, \cdots, 9\}$ 上の文字列を定数とする．この時，以下の BNF 記法によって，式 (expression) の集合を帰納的に定義することができる．

$$\langle\, \text{式} \,\rangle \quad ::= \quad \langle\, \text{変数} \,\rangle \mid \langle\, \text{定数} \,\rangle \mid \langle\, \text{式} \,\rangle\texttt{+}\langle\, \text{式} \,\rangle \mid \langle\, \text{式} \,\rangle\texttt{-}\langle\, \text{式} \,\rangle$$

$\langle$ 式 $\rangle$ や $\langle$ 変数 $\rangle$ は，それぞれ，式の集合の要素，変数の集合の要素を表す．縦棒 | は，帰納的定義の定義節 (basis や induction step) の区切りを表す．上記の BNF は以下の帰納的定義を表している．

- 変数は式である．
- 定数は式である．
- x が式で y が式ならば x+y は式である．
- x が式で y が式ならば x-y は式である．

たとえば，abc+231-50 が式となる．

BNF の例をもう 1 つ与える．

$$\begin{aligned}
\langle\, \text{文} \,\rangle \quad &::= \quad \langle\, \text{変数} \,\rangle\texttt{:=}\langle\, \text{式} \,\rangle \mid \texttt{begin}\ \langle\, \text{文の列} \,\rangle\ \texttt{end} \\
&\qquad\quad \mid \texttt{while}\ \langle\, \text{式} \,\rangle\ \texttt{do}\ \langle\, \text{文} \,\rangle \mid \texttt{if}\ \langle\, \text{式} \,\rangle\ \texttt{then}\ \langle\, \text{文} \,\rangle\ \texttt{else}\ \langle\, \text{文} \,\rangle \\
\langle\, \text{文の列} \,\rangle \quad &::= \quad \langle\, \text{文} \,\rangle \mid \langle\, \text{文の列} \,\rangle\texttt{;}\langle\, \text{文} \,\rangle
\end{aligned}$$

これは while プログラムと呼ばれる簡単なプログラム言語の文 (statement) の定義を与えるものである．ここで,「文」と「文の列」という 2 つの集合を同時に帰納的に定義している．たとえば，以下のものが「文の列」になる．

$$x\texttt{:=}3;\ y\texttt{:=}0;\ \texttt{while}\ x\ \texttt{do begin}\ y\texttt{:=}y\texttt{+}3; x\texttt{:=}x\texttt{-}1\ \texttt{end}$$

4.2 帰納的に定義された関数

前節では，リストを操作する部分関数 head, tail を用いたが，これらは非形式的に意味を与えただけであり，厳密な定義は与えていなかった．本節では，リストや木など帰納的に定義された集合を定義域とする関数 (または部分関数) を定義する方法を与える．

定義 4 [関数 $f : A \to B$，ただし A は帰納的に定義された集合]

- (basis) A の帰納的定義の basis で与えた要素 x に対して，$f(x)$ の値を B の要素となるように定める．
- (induction step) A の帰納的定義の induction step が「$a \in A$ から $b \in A$ を作る操作」であったとすると，$f(a)$ の値を用いた式で $f(b) \in B$ の値を定める．

集合 A の帰納的定義に曖昧さがなければ，このように定義することにより，f は関数となることが保証される．すなわち，$f : A \to B$ である．

自然数の集合 $\mathcal{N}$ に対して，$f : \mathcal{N} \to B$ となる関数 f は次のように帰納的に定義される．

- $0 \in \mathcal{N}$ に対して $f(0)$ の値を定める．
- $n \in \mathcal{N}$ に対して，$f(n)$ の値を用いて，$f(n+1)$ の値を定める．

このことを，以下のように表記する．

$$f(n) = \begin{cases} \cdots & (n = 0 \text{ の時}) \\ \cdots f(m) \cdots & (n = m+1 \text{ の時}) \end{cases}$$

ここで，$n = m+1$ のときに $f(m)$ 以外の f の値を使ってはいけない (後述する拡張の場合を除く)．たとえば，$f(n+1)$ の値を使って

$$f(n) = \begin{cases} 0 & (n = 0 \text{ の時}) \\ f(n+1) + 3 & (n = m+1 \text{ の時}) \end{cases}$$

としたものは，帰納的定義にならない [3]．

例 95 $f : \mathcal{N} \to \mathcal{N}, n \in \mathcal{N}$ に対して 1 から $2n+1$ までの奇数の和を求める関数 $f(n)$ を定義しよう．非形式的に考えると，$n \geq 1$ ならば

$$\begin{aligned} f(n) &= 1 + 3 + \cdots + (2(n-1)+1) + (2n+1) \\ &= f(n-1) + (2n+1) \end{aligned}$$

である．これをもとにして，以下の帰納的定義で f を定義できる．

$$f(n) = \begin{cases} 1 & (n = 0 \text{ の時}) \\ f(m) + 2n + 1 & (n = m+1 \text{ の時}) \end{cases}$$

例 96 加算 $\mathrm{add} : \mathcal{N} \times \mathcal{N} \to \mathcal{N}$.

加算は引数が 2 つあり，両方とも帰納的に定義された集合 $\mathcal{N}$ の要素であるので，加算を 2 通りに定義することができる．

$$\mathrm{add}(n, m) = \begin{cases} m & (n = 0 \text{ の時}) \\ \mathrm{add}(p, m) + 1 & (n = p+1 \text{ の時}) \end{cases}$$

$$\mathrm{add}(n, m) = \begin{cases} n & (m = 0 \text{ の時}) \\ \mathrm{add}(n, p) + 1 & (m = p+1 \text{ の時}) \end{cases}$$

厳密に言うと，加算の定義の右辺に現れる $+1$ は加算ではなく，自然数の帰納的定義に現れる $+1$ (ある自然数から次の自然数を構成する操作) である．

[3] 多くのプログラミング言語では $f(n) = \cdots f(n+1) \cdots$ のように f を定義する式の右辺で f を自由に使ってよい．すなわち，再帰的関数 (recursive function) を自由に定義して使うことが許されている．一方，本章で述べている「帰納的に定義された関数」(inductively defined function) は，再帰的関数の一種であるが，右辺で使ってよい f の値が制限されている点が異なる．

例 97 乗算 $\mathrm{times} : \mathcal{N} \times \mathcal{N} \to \mathcal{N}$.

乗算も同様に 2 通りの定義が考えられるが，ここでは 1 つのみ与える．

$$\mathrm{times}(n, m) = \begin{cases} 0 & (n = 0 \text{ の時}) \\ \mathrm{times}(p, m) + m & (n = p + 1 \text{ の時}) \end{cases}$$

例 98 階乗 $\mathrm{factorial}(n)$.

$$\mathrm{factorial}(n) = \begin{cases} 1 & (n = 0 \text{ の時}) \\ n \times \mathrm{factorial}(m) & (n = m + 1 \text{ の時}) \end{cases}$$

例 99 引数の長さに応じたリストを生成する関数 $f : \mathcal{N} \to List_{\mathcal{N}}$ で $f(n) = \langle n, \ldots, 1, 0 \rangle$ となるもの．

$$f(n) = \begin{cases} \langle 0 \rangle & (n = 0 \text{ の時}) \\ \mathrm{cons}(n, f(p)) & (n = p + 1 \text{ の時}) \end{cases}$$

$$\begin{aligned} f(3) &= \mathrm{cons}(3, f(2)) \\ &= \mathrm{cons}(3, \mathrm{cons}(2, f(1))) \\ &= \mathrm{cons}(3, \mathrm{cons}(2, \mathrm{cons}(1, f(0)))) \\ &= \mathrm{cons}(3, \mathrm{cons}(2, \mathrm{cons}(1, \langle 0 \rangle))) \\ &= \langle 3, 2, 1, 0 \rangle \end{aligned}$$

ここまでの定義では，$n = 0$ の場合と $n \geq 1$ の場合に分けたが，その変形・拡張として $n = 0, 1, \cdots, k$ の場合と $n > k$ の場合に分ける等が考えられる．この場合，$f(n)$ の定義において $m < n$ となる $f(m)$ の値を使ってもよい．

例 100 Fibonacci 数列 $\mathrm{fib}(n)$ は以下のように帰納的に定義される．

$$\mathrm{fib}(n) = \begin{cases} 1 & (n = 1, 2 \text{ の時}) \\ \mathrm{fib}(n-1) + \mathrm{fib}(n-2) & (n > 2 \text{ の時}) \end{cases}$$

例 101 リストを操作する部分関数 $\mathrm{head}; List_A \to A$, $\mathrm{tail}; List_A \to List_A$.

$$\mathrm{head}(x) = \begin{cases} \text{未定義} & (x = \langle\rangle \text{ の時}) \\ y & (x = \mathrm{cons}(y, L) \text{ の時}) \end{cases}$$

$$\mathrm{tail}(x) = \begin{cases} \text{未定義} & (x = \langle\rangle \text{ の時}) \\ L & (x = \mathrm{cons}(y, L) \text{ の時}) \end{cases}$$

例 102 リストの長さ $\mathrm{length} : List_A \to \mathcal{N}$.

$$\mathrm{length}(x) = \begin{cases} 0 & (x = \langle\rangle \text{ の時}) \\ 1 + \mathrm{length}(L) & (x = \mathrm{cons}(y, L) \text{ の時}) \end{cases}$$

この定義の 2 行目は，$1 + \mathrm{length}(\mathrm{tail}(x))$ と書いても同じである．

例 103 リストの連結 (concatenation) $\oplus : List_A \times List_A \to List_A$.

$$x \oplus y = \begin{cases} y & (x = \langle\rangle \text{ の時}) \\ \mathrm{cons}(z, L \oplus y) & (x = \mathrm{cons}(z, L) \text{ の時}) \end{cases}$$

例 104 2 分木の節の数を数える関数 $\mathrm{nodes} : \mathrm{BinTree}_A \to \mathcal{N}$.

$$\mathrm{nodes}(x) = \begin{cases} 0 & (x = \langle\rangle \text{ の時}) \\ 1 + \mathrm{nodes}(L) + \mathrm{nodes}(R) & (x = \mathrm{Tree}(L, y, R) \text{ の時}) \end{cases}$$

例 105 2 つの文字列を結合する関数 $\cdot : \Sigma^* \times \Sigma^* \to \Sigma^*$. ここでは，$\Sigma^*$ は「文字列の定義 1」により定義されているとする．

$$s \cdot t = \begin{cases} t & (s = \Lambda \text{の時}) \\ x\ (u \cdot t) & (s = xu \text{ の時}) \end{cases}$$

この定義で，$s, t, u \in \Sigma^*$ であり $x \in \Sigma$ である．「文字列の定義 1」では，$x \in \Sigma$ と $u \in \Sigma^*$ から $s = xu$ となる文字列 s を構成したので，それに対する関数も上記の形を取る．もし，「文字列の定義 2」を採用した場合は，以下のように定義すべきである．

$$s \cdot t = \begin{cases} s & (t = \Lambda \text{の時}) \\ (s \cdot u)\ x & (t = ux \text{ の時}) \end{cases}$$

4.3 帰納法による証明

帰納的に定義された集合 A に対して，A のすべての要素に対してある性質 P が成立することを証明するための方法が帰納法 (induction) である．

4.3.1 数学的帰納法

自然数に対する帰納法が数学的帰納法である．すべての自然数 x に対して, $P(x)$ が成り立つことを示すために，以下の 2 つのことを示せばよい．

- (base case) $P(0)$ が成り立つことを示す.
- (induction step) どんな $x \in \mathcal{N}$ に対しても，$P(x)$ が成り立つことを仮定[4] して，$P(x+1)$ が成り立つことを示す.

例 106 定理 「0 から n までの数の平方 (2 乗) の和は $\frac{n(n+1)(2n+1)}{6}$ である」
証明: $P(n)$ を上記定理中の「...」という命題とする．

- $P(0)$ は「$0^2 = \frac{0 \cdot 1 \cdot 1}{6}$」となるので正しい．
- $P(n)$ が正しいと仮定する．0 から $n+1$ までの数の平方の和は

$$\frac{n(n+1)(2n+1)}{6} + (n+1)^2 \quad = \quad \frac{(n+1)(n+2)(2n+3)}{6}$$

  となるので，$P(n+1)$ も正しい．
- 以上より，どんな $n \in \mathcal{N}$ に対しても $P(n)$ は正しい．

基本的な数学的帰納法は，$\forall n \in \mathcal{N}.P(n)$ を示すためのものであったが，様々な形に拡張して利用されることが多い．

[4] この仮定のことを帰納法の仮定 (induction hypothesis) という.

- n が動く範囲が 0 から始まらない時:

  $n \geq k$ なる任意の自然数に対して $P(n)$ が正しいことを示すためには，base case として $P(k)$ を示し，induction step として $n > k$ なる任意の自然数 n に対して，$P(n)$ を仮定して $P(n+1)$ が成立することを示せばよい.

- n が動く範囲が飛び飛びの時:

  任意の正の奇数に対して $P(n)$ が正しいことを示すためには，base case を $n = 1$ とし，induction step では，$P(2k+1)$ を仮定して $P(2k+3)$ を示せばよい.

- 帰納法の仮定を強めたい時:

  induction step では，$P(n-1)$ だけでなく $0 \leq k < n$ なる任意の k に対して $P(k)$ が成り立つことを仮定して $P(n)$ を証明してもよい.

例 107 関数 fib と任意の正の整数 $n > 0$ に対して，

$$fib(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

が成り立つことを証明する.

与えられた式の右辺を $g(n)$ とおく．すなわち，

$$g(n) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right)$$

である.

正の整数 n に関する数学的帰納法を使って $\forall n.(n > 0 \Rightarrow fib(n) = g(n))$ を示す.

- $n = 1$ のとき

$$g(1) = \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right) - \left(\frac{1-\sqrt{5}}{2}\right)\right) = 1 = fib(1)$$

- $n = 2$ のとき

$$g(2) = \frac{1}{\sqrt{5}}\left(\left(\frac{6+2\sqrt{5}}{4}\right) - \left(\frac{6-2\sqrt{5}}{4}\right)\right) = 1 = fib(2)$$

- $n > 2$ のとき

  $k < n$ となる任意の正整数 k に対して $fib(k) = g(k)$ が成り立つと仮定する. $n-2$ と $n-1$ は正整数だから $k = n-2$ と $k = n-1$ に対してもこの式が成立する．このとき，

$$\begin{aligned}
fib(n) &= fib(n-2) + fib(n-1) \\
&= g(n-2) + g(n-1) \\
&= \frac{1}{\sqrt{5}}\left(\left(\frac{3+\sqrt{5}}{2}\right)\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \left(\frac{3-\sqrt{5}}{2}\right)\left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\right) \\
&= \frac{1}{\sqrt{5}}\left(\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right) \\
&= g(n)
\end{aligned}$$

4.3.2 リストに関する帰納法

リストに対する帰納法は，以下の形を取る．すべての $L \in List_A$ に対して, $P(L)$ が成り立つことを示すには以下の 2 つを証明すればよい.

- $P(\langle\rangle)$ が成り立つを示す.
- 任意の $L \in List_A$ と任意の $a \in A$ に対して，$P(L)$ が成り立つことを仮定して $P(a::L)$ が成り立つことを示す.

例 108 任意の $x, y \in List_A$ に対して以下の等式が成立することを証明する．

$$\mathrm{length}(x \oplus y) = \mathrm{length}(x) + \mathrm{length}(y)$$

(証明) リスト x に関する帰納法を使う．つまり，全ての $x \in List_A$ に対して,「$\forall y \in List_A.\mathrm{length}(x \oplus y) = \mathrm{length}(x) + \mathrm{length}(y)$ 」を証明する [5]．この「...」を命題 $P(x)$ とする．

- (base case) $x = \langle\rangle$ の時. 任意の $y \in List_A$ を取る．

  $$\mathrm{length}(\langle\rangle \oplus y) = \mathrm{length}(y) = \mathrm{length}(\langle\rangle) + \mathrm{length}(y)$$

  となって，命題 $P(x)$ は成立する．

- (induction step) $x = a :: z$ の時. 任意の $y \in List_A$ を取る．

  $$\begin{aligned}
  \text{(左辺)}\ \mathrm{length}((a :: z) \oplus y) &= \mathrm{length}(a :: (z \oplus y))\ (\oplus \text{ の定義による}) \\
  &= \mathrm{length}(z \oplus y) + 1\ (\text{length の定義による}) \\
  &= \mathrm{length}(z) + \mathrm{length}(y) + 1\ (\text{帰納法の仮定，すなわち } P(z) \text{ による})
  \end{aligned}$$

  $$\text{(右辺)}\ \mathrm{length}(a :: z) + \mathrm{length}(y) = \mathrm{length}(z) + 1 + \mathrm{length}(y)\ (\text{length の定義による})$$

  となって，命題 $P(a :: z)$ は成立する．

  以上より，全ての $x \in List_A$ に対して $P(x)$ は成立する．

帰納法による証明は，似て非なる式がいくつも出てくるので，どの定義の形にあてはめて変形を行っているかを明示的に書くようにするとよい．たとえば，帰納法で証明したい命題と，帰納法の仮定は，命題の形そのものは全く同じで，対象となるデータが少し違うだけであり，間違えやすい [6]．上記の証明では，等式変形のたびに，その根拠を右端に記述した．このような習慣をつけることにより，間違いを減らすことができ，また，あとで証明の正しさをチェックしやすくなる．

[5] このように，証明したい式に x と y という 2 つのリストが現れているときは，x に関する帰納法を使う場合と y に関する帰納法を使う場合がある．どちらを使うと，うまく行くかは問題によるので人間が考える必要がある．

[6] よくある間違いは，証明したい論理式を仮定してしまうというものである．

4.3.3　２分木に関する帰納法 (†)

すべての $x \in \mathrm{BinTree}_A$ に対して, $P(x)$ が成り立つことを示すには以下の 2 つを言えばよい.

- $P(\circ)$ が成り立つを示す.
- 任意の $x, y \in \mathrm{BinTree}_A$ と，任意の $a \in A$ に対して，$P(x)$, $P(y)$ が成り立つことを仮定して，$P(\mathrm{Tree}(x, a, y))$ が成り立つことを示す.

ここまでに述べた自然数，リスト，木に対する帰納法はすべて，帰納的定義の構造に沿ったものであるため，構造帰納法 (structural induction) と呼ばれる.