プログラミングは数学教師を夢見るか

小泉真也、濱田百代

● **要約**

プログラミングの土台は情報科学であり、情報科学の土台は数学である。数学の視座に立つ者と情報学の視座に立つ者それぞれが、斯様な関係を理解しながら互いの「スマートさ」の差異を「なんとなく」実感している。

筆者らの関心は、プログラミング教育の必修化が迫る中、その対象として数学が適するかにある。プログラミング・ソースが「数学の解説者」たりうるかを議論するにあたっては、それ以前の問題として「プログラミング」と「数学」の差異を具体的に追及することが必要となるであろう。本稿では、プログラミングや数学、そしてそれぞれの教育に関する事例を四散的に取り上げることによって、両者の違いや、数学科におけるプログラミング教育の可能性を考察した。

● **キーワード**

プログラミング教育
数理的（数学的）思考
算法的思考

1. はじめに

古くから「読み・書き・そろばん」といわれてきたように、計算は近世末期以降、初等教育で獲得させる基礎的な能力･学力のひとつである。いまやコンピュータは、我々の日常生活に欠くことのできない道具となっており、アプリケーション・ソフトウェアの充実によって、「読み・書き・そろばん」のそれぞれに、コンピュータが介在するまでになっている。

文部科学省が検討を進めている、2020 年からの小学校でのプログラミング教育必修化は、この基礎的な能力・学力の中にプログラミングが位置付けられることを意味しており、現時点でも、初等中等教育段階におけるプログラミング教育のさまざまな実例も紹介されているが[1]、生活科、技術科、情報科を網羅する一方で、算数・数学科を対象とした事例が皆無という現状が確認できる。

機能がそうであり、名称がそうであるが、コンピュータは「計算機」である。小泉は、かねてより、プログラミング教育を数学科に適用するという提案を行っている[2]。一連の提案は、社会的要請が高まりつつある「プログラミング人材」の育成を意識したものであり、R などの解析用ソフトウェアを利用したデータ処理の習得を念頭に置いている。また、プログラミング言語で記述したアルゴリズムは、従来の「紙と筆記具」で学んできたものとは異なる解法を提示するものであり、数学を得意とする生徒・学生には数学の多様な考え方を学び、数学を苦手とする生徒・学生には数学を「操る」手段を提供することを狙いとしている。

プログラミングを学ぶことによって、数学を得意とする生徒・学生には数学の多様な考え方を学ぶことができ、数学を苦手とする生徒･学生には数学を「操る」手段を得ることができるとするならば、その根拠として、典型的な数学の考え方と、アルゴリズムに基づくーすなわち計算機科学的な数学の考え方は、性質を異にすることを示す必要がある。本稿では、その 2 つの考え方を比較することによって、具体的にどのような違いがあるのかを示す。

2. 「抽象化」のアプローチとその理解

アレクサンドルフ（A.D. Aleksandrov）は、数学の特徴として、次の性質を挙げている[3]：

- 色々な段階の抽象化を持つ抽象性
- 精密性と段階的な厳密性
- 量的な関係
- 広大な応用範囲

すなわち、数学は抽象化の学問といえる。広大な学問領域である「数学」の中で、計算機科学の及ぶ範囲はおそらく極一部であろう。これら数学の特徴は、計算機科学の特徴でもあるようだが、その様式は数学の特徴とは異なるものとして理解されがちである。だが、それとて「計算機による計算」のために、従前の数学の概念によって数学自らを拡張しているに過ぎない。

現に計算機科学における「抽象化」は、数学の抽象化のアナロジーである。たとえば「数」や「数の計算の概念」は、数学も計算機科学もほぼ共通した概念である。アナロジーを逸脱しない範囲で、計算機科学に独特の概念を挙げるならば「制御抽象化」および「データ抽象化」がこれにあたるだろう。「制御抽象化」とは、サブプログラムや制御構造など、処理動作の抽象化である。たとえば、実行プロセスの移動は論理を土台としており、反復制御は数学的帰納法を土台としている。「データ抽象

化」とは、あらゆるデータを有限のビット列で表現する、データ構造の抽象化である。たとえば、データ型の概念や、減算が「桁あふれを利用した 2 の補数の加算」で行われることは、コンピュータの物理構造に起因した数値表現の制約である。文字はコンピュータにおいて尺度水準として扱われるが 'A' + ' ' = 'a' のような計算を許容するのは、ある意味コンピュータの都合である。

数学においても、プログラミングにおいても、この「抽象的にとらえる」ことが、初学者にとっては真新しい概念であり、違和感を覚えることがあるかもしれない。では、「抽象化」を苦もなく受け入れる者は、生まれつき「数学脳」のようなものを備えているかと問われれば、そうでもないようである。バターワースによれば[4]、ヒトは生来、4〜5 程度の数の多少を認識する能力を有しているという。その能力は文化が生み出した「概念を拡張するツール」を獲得することによって計数能力へと発達し、計数能力は経験によって高まるとしている。

そもそも「わかる」、「できる」とはどういうことだろうか。新井は数学科教育を「定義から出発して、正しく論理推論をして、結論を出せるような力を育てる『はずの』科目」と位置付けている。しかしながら、現実は例題に似せて問題を解いており、諸概念に対して具体例を挙げることはできるが、定義が定着していないといった状況が散見されることを問題に挙げている[5]。つまり「問題を解くことは『できる』が、答えに至る筋道は『わからない』」ということであり、数学科教育において思考の外化を徹底するための時間が割きにくい状況が伺える。いわゆる「つまずき」についてはその要因が教え子に帰するか、それとも教師に帰するか、しばしば議論を呼ぶが、ここでは責任の所在を意図するものではない。

銀林[6]の表現を借りれば、「わかる」とは以下の 2 つに区別できる：

- やりかたがわかる（アルゴリズムの把握）
- わけがわかる（意味の理解）

この 2 つは必ずしも一致しない。これを逆説的に述べれば、「わからない」とは、以下の 3 つに分類できる：

- やりかたも意味もわからない
- やりかたはわかるが、意味がわからない
- 意味は分かるが、やりかたがわからない

プログラミングでは「やりかた」を記述する。少なくとも「やりかた」がわかれば、問題は解決「できる」。

国立情報学研究所（大学共同利用機関法人 情報・システム研究機構）を中心に 2011 年から続くプロジェクト「ロボットは東大に入れるか (http://21robot.org/)」において研究・開発が進められた人工知能「東ロボくん」は、2021 年度の東京大学合格を目標としてきた。研究の成果として、2015 年度には、国公立 33 大学、私立 441 大学で合格可能性 80%以上を獲得するなど、順調に成績を伸ばしてきた。しかしながら、以降の成績が伸び悩んだことで、2016 年 11 月、目標達成は困難であるとし、研究方針を転換したことは記憶に新しい。

現在のコンピュータは、ただ記憶と計算と検索のみで「考える」を模倣する。ゆえに東ロボくんも例にもれず、蓄積した知識や論理を扱う問題を得意とするが、読解力に相当する意味解析を伴う問題

を苦手とするという。

いささか極論ではあるが、コンピュータの高速な処理は大量の情報を記憶するために十分な時間を保証するものであり、記憶を忘れることなく、高速に取り出すことができるならば、ヒトでも東ロボくんと同等の成績は可能であり、さらには読解力によってヒトが東ロボくんより良い成績を獲得することができる。

戸塚[7]は、マーヴィン・ミンスキーが唱えた「パパートの原理（Paperts' principle）」[8], [9] をもとに、「学習」を「積み木細工のような"知識の塔"を築く活動」ではなく、「知識のリンクによるネットワークをつくること」と述べている。学校における算数・数学の指導法は、おそらく積み木細工のように知識を積み上げる直列的なものであり、途中でひとつでもわからない知識があれば、これがつまずきのきっかけを生む。これに対して、ネットワーク化した知識は、知識の関連付けこそが「学び」であり、わからない知識を別の知識で補いながら、より複雑に拡大していくとし、これを「発達」と称している。人工知能の学習もこの考え方に基づくものであり、あらゆる知識の組み立てから、正解という「報酬」を得る筋道を探索し、最も効率の良い手法を残すというアプローチをとっている。

では、なぜ大部分のヒトは人工知能よりも「できない」のか。やる気の有無を別とすれば「できない」理由は、おおよそ以下の 3 つが考えられる：

- 人手が足りない（ひとりではできない）
- 環境が整っていない（このままではできない）
- 時間が足りない（すぐにはできない）

教える人手の多少や、環境の差に関わらず、「できない」生徒・学生は一定の割合で現れるとするなら、残るは「できるまでやるだけの時間が充分に確保できない」ということになる。

抽象化の一例として、数学の数学記号や、プログラミング言語のおよそ自然言語とはかけ離れたキーワードが、学習を難しくしているという側面はあるだろう。しかしながら、数学記号も、プログラミング言語も、日常言語を鈍化したものであり、それぞれ学習のためには、ある段階でその「文法」を学ぶ時間を確保することでしか、「できる」ようになる対策はないと考える。

3. 「算法」とは

コンピュータによる組版システム TEX の開発者であるドナルド・クヌース（Donald Ervin Knuth）は、数学者であり、また計算機科学者でもある。特に、コンピュータプログラミングに関する著作 "The Art of Computer Programming" 等の業績は、計算機科学分野の発展に大きく貢献した。

クヌースは、いわゆる計算機科学を「算法（Algorithmics）」の研究と位置付けている。クヌースがいう「算法」とは、特定の問題を解くための方法にとどまらず、正しく定義された過程を取り扱う概念全範囲を指しており、操作を施すべきデータ構造も考慮している。

クヌースは、心理学者デ・ヤング（Gerrit De Young）の実験を挙げ、「算法的思考」の存在を意識させる発言を行っている。これは、プログラミングを含む計算機科学の入門講座を受講した学生を、計算機科学専攻の 135 名と、社会科学専攻の 35 名に分けて、A…数量的な能力、B…学生自身のプログラミング能力に対する自己採点、C…プログラミング能力に対する教師の評価について、A, B, C の関係を調査したものである。どちらのグループも B と C は相関係数でおよそ 0.6 を示しており、C にお

ける教師の評価を信用に足るものとした。そのうえで、A と B、および A と C の相関について、計算機科学専攻のグループには全く相関が見られず、一方で社会科学専攻のグループには相関係数でおよそ 0.4 を示した、というものである。

クヌースは、さらに、ランダムに選んだ 9 冊の数学書の 100 ページ目に書かれた主題の最初の結果を観察することによって、暫定的な結論として、「『数学的な考え方』といった単一の孤立した概念のようなものは元々存在しない」ことを確認している。

ただし、計算機科学の視座から、数学には 2 つの思考型が欠如していると指摘した。ひとつは「操作の節約」といった概念がほとんどないことである。もうひとつは「過程の状態に関する動的な概念」であり。クヌースはこれを「スナップショット」と呼んでいる[10], [11]。

典型的な数学に対して、計算機科学における「操作の節約」や「スナップショット」のような概念の拡張が起こった背景は、コンピュータのアーキテクチャに起因するところが大きい。

図 1 は、今日、実用的なコンピュータ・アーキテクチャであるノイマン型コンピュータについて、データの記憶と演算のみに着目した概略図である。数値演算は CPU（中央処理装置）とメイン・メモリ（主記憶装置）のデータのやり取りによって行い、矢印はデータの流れを示す。

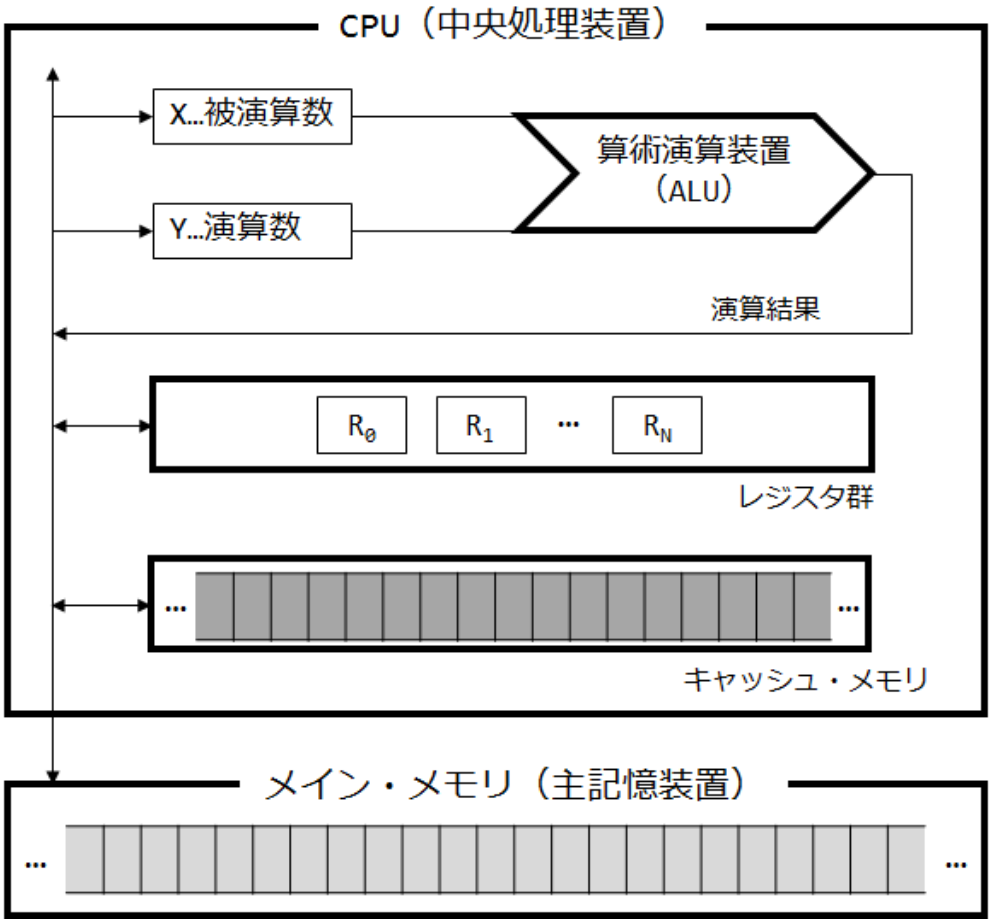


図 1 ノイマン型コンピュータの概略図

データの記憶は、用途に応じてメイン・メモリ、キャッシュ・メモリ、レジスタの 3 つに分類される。メイン・メモリは演算を構成する概念のメタファであり、処理が終了するまで演算に用いるすべてのデータを格納する。キャッシュ・メモリは暗記のメタファであり、使用頻度の高いデータを蓄積することによってメイン・メモリへのアクセスを減らすことができ、処理を高速化することができる。レジスタは、想起のメタファであり、CPU 内部のたかだか数個の高速な記憶装置である。これらの記憶領域に値を格納することは、マス目に数値を書くことに似ている。コンピュータで扱う数値の最小単位は 0 か 1 かのビット（bit）であるが、マス目のサイズはバイト（byte）というビット列に基づいており、文字、整数、実数のそれぞれで占有するバイト数が異なる。マス目は有限個の線形構造であり、一度書いた値は、新たな値で上書きするまで消えることはない。

演算処理においては、必要なデータだけが、メモリ、またはキャッシュ・メモリからいずれかのレジスタに読み込まれる。算術演算装置（ALU）は、2 つの値を取り込む仕組みである。X に取り込む「被演算数」は必ずレジスタから取得し、Y に取り込む「演算数」はメモリ、キャッシュ・メモリ、レジスタのいずれからでも読み込むことができる。

ALU の演算方式は通常「2 アドレス式」と呼ばれ、多くのプログラミング言語で

$$X = X + Y$$

といった記述をするように、2 値の演算結果は X が取得したレジスタに返る。上記の演算は、厳密には $X \leftarrow X + Y$ （現在の X と Y の和を、新しい X として更新する）と記述すべきであろう。これを「指定演算」と呼ぶことがある。指定演算によって被演算数は「破壊」されるため、事前に他のレジスタに複製するか、事後にメイン・メモリから再度取得することが必要となる。コンピュータの演算が 2 アドレス式を採用したのは、CPU の命令処理の効率化を考慮したためである。レジスタに返った演算結果は、処理の終了時にメイン・メモリの所定の位置に書き込まれ、処理結果はメイン・メモリから読み出す。

コンピュータの計算は値を記述する空間に制約があり、従来「紙と筆記具」でできていた手法の実装が困難である。計算機科学の視座における独特の思考は、斯様な制約を克服するための発想の賜物といえる。

4. プログラミング教育の利点

プログラミング言語を学ぶことによる効果として、しばしば「論理的な思考が身につく」といった言及がなされる。しかしながら、その多くは「どのような論理か」といった視点に欠けることが多く、確信的、確証的なものとは言い難い。

また、コーディング（プログラムを書くこと）よりも、論理的な思考を学ぶことが重要であるといった言及もしばしば見られるが、コンピュータを動かすためにはコーディングを避けて通ることができない。特に数学科でプログラミングを扱う場合、データ型や、メモリのアドレス、ロー・コンテキストな言語体系は、解法を把握する上での妨げとなるであろう。

プログラミングでは、おおよそ以下の手順で問題を解決する：

1. 問題から、ルールや規則性といった構成要素をとらえる
2. 入出力の対応や終了時の結果など、適切な推論を行う

3. コンピュータが処理できる問題に単純化する
4. プログラミング言語を用いて処理を記述する
5. 論理の正誤ー意図通りに動くかーを検証する

大抵の学習では、結果の正誤はわかるが、紙などに逐一書き出さない限り、誤りの経緯や自身の想定を検証することは難しい。またプログラミング言語の構文を誤るとプログラム自体が動作しない。一連の手順は、原因と結果を厳密に結びつける行為といえる。

プログラミング教育の特色は「コンピュータに教える」ことにあり、他のコンピュータ支援教育（CAI）のような「コンピュータから教わる」考え方とは一線を画す。ここで、ヒトは、コンピュータの構造がもたらす計算環境のもと、思考しないコンピュータが解きやすい方法は何か、その探索に知恵を絞ることを迫られる。プログラミング・ソースが残ることで、その思考は外化され、思考を共有するためには他者が判読しやすいコーディングが求められる。

5. 問題の見方

3 章にて、「数学的な考え方」といった単一の孤立した概念のようなものは元々存在しない一方で、「算法的な考え方」という拡張的な概念の存在があることを述べた。本稿では計算機科学の位置づけを、数学の一部と述べており、「数学全体」と混同無きよう、「数学的な考え方」という呼び方は改める必要があるであろう。以降、「算法的ではない考え方」を便宜的に「数理的な考え方」と呼ぶこととする。

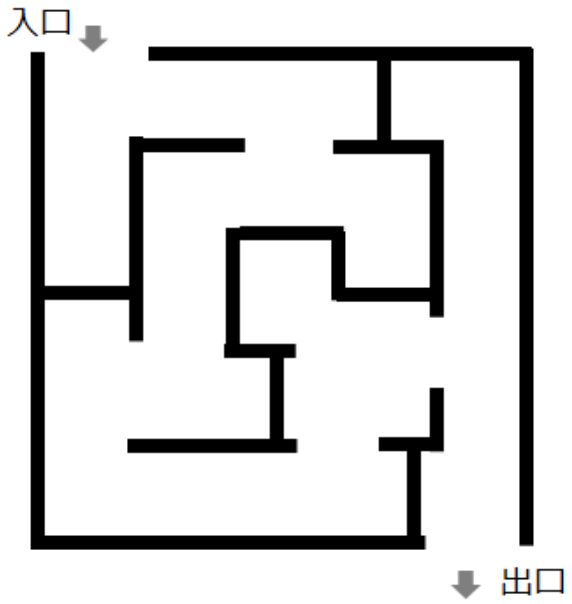
クヌースが "Algorithmics" と称したように、算法的な考え方は「系列的」な見方に基づくといえる。数理的な考え方が算法的な考え方と異なるとすれば、数理的な考え方は「俯瞰的」な見方であると仮定しよう。これをふまえて、いくつかの例における「見方の違い」を示す。

5.1. 迷路

図 2 に簡単な迷路の例を示す。(a)は迷路の俯瞰図であり、(b)は迷路の「抜け方」を文書化したものである。

いま、A…迷路を俯瞰する者と、B…迷路の入口にいる者がいるとしよう。A に比べれば、B は迷路の脱出経路を素早く解決できる状況にある。迷路を俯瞰するだけなら、敢えて脱出経路を外化する必要はないが、図 2(b)のように外化することによって B は迷路を俯瞰せずとも容易に迷路を脱出できる手段を得る。では、図 2 の(a)→ (b)のような変換を経なければ、B は経路を見出すことができないかと言えばそうではない。行き止まりがあれば戻るということを繰り返すことで、力任せに経路を探索することも可能である。このアプローチは「バックトラッキング」という、れっきとしたアルゴリズムである。

図 2(b)はプログラミング・ソースに相当し、思考を共有するツールとなる。この成果は俯瞰的な見方によって効率よく導くことができ、系列的な見方のみでも力任せに導くことができる。



1. 左に90度回転
2. 右側にある分岐口まで直進
3. 分岐口に到達したら、右に90度回転
4. 分岐口を入り、右に90度回転
5. 行き止まりまで直進
6. 行き止まったら、左に90度回転
7. 右側にある分岐口まで直進
8. 分岐口に到達したら、右に90度回転
9. 行き止まりまで直進
10. 行き止まったら、左に90度回転
11. 行き止まりまで直進
12. 行き止まったら、左に90度回転
13. 左側にある分岐口まで直進
14. 分岐口に到達したら、左に90度回転
15. 右側にある分岐口まで直進
16. 分岐口に到達したら、右に90度回転
17. 行き止まりまで直進
18. 行き止まったら、右に90度回転
19. 直進

(a) 俯瞰図　　(b) 迷路(a)の「抜け方」

図 2 迷路

5.2. 円

図 3 に数式による円の表現を示し、図 4 に数式に基づく C 言語のプログラミング・ソースを示す。

半径 r の円について

- 関数：　$x^2 + y^2 = r^2$
  ただし、 $x = r\cos\theta, y = r\sin\theta$
- 円周長 l：　$l = 2\pi r$
- 面積 s：　$s = \pi r^2$

図 3 数式による円の表現

数式が示すのはこの世の中の「すべて（all）」の円である。一方でプログラミング・ソースは、具体的な (r, θ) の入力を要求し、関数上のただ一つの点と、r に対応する円周長および面積を出力する。プログラムが処理する値の「それぞれ（every）」は、集約によって公理を意識づける。

また、数式は「すべて」を我々の視界の及ぶ空間に「凝縮」して表現しようとしている。対してプログラミングによる「それぞれ」の求め方は、主プログラムたる main(){} の空間に「捨象」的に表現され、抽象化されたキーワードは、主プログラムの外部に副プログラムとして表されている。

5.3. 四捨五入

数理的な考えは公理に基づく。たとえば我々は、小数点以下第一位での四捨五入を「処理する桁の値が 0.5 以上なら切り上げ、0.5 未満なら切り捨て」と理解している。一方で、このアルゴリズムは条件分岐などを用いることなく「処理する値に 0.5 を加えて小数点以下を切り捨てる」ことで効率的に記述できる。両者の考え方は可逆的に結びつくはずであるが、アルゴリズムから公理を導くことは容易ではないだろう。

```
#include <stdio.h>
#include <math.h>
#define   pi 3.14   //pi は円周率

double x-axis(double r, double rad)
{
          return    r * cos(rad);
}

double y-axis(double r, double rad)
{
          return    r * sin(rad);
}

double length(double r)
{
          return    2 * pi * r;
}

double area(double r)
{
          return    pi * pow(r, 2.0);
}

int main()
{
          double    radius;   //半径
          double    deg;      //角度

          scanf("%f, %f", radius, deg);
                    //半径と角度を入力
          printf("x... %f", x-axis(radius, deg));
                    //x座標値を出力
          printf("y... %f", y-axis(radius, deg));
                    //y座標値を出力
          printf("Length... %f", length(radius));
                    //円周長を出力
          printf("Area... %f", area(radius));
                    //円の面積を出力

          return    0;
}
```

図４ 図３に基づくアルゴリズム

5.4. 数学の教科書

図 5 は、微分・積分を中心に扱う数学 III の教科書の目次である。紙の書籍は系列的である。我々は「教科書の順序で」算数を、そして数学を学んできた。

図 5 比較的「系列的な」教科書ー数学 III の教科書の目次

ここでパパートの原理を思い返してみよう。もし微分・積分が系列的な学びによって身につくのだとしたら、「『極限』が理解できなければ『微分』は理解できない」ということであろうか（図 6）。確かに、微分・積分は関数の問題であり、関数の知識は必要であろう。たとえば微分を「関数の変化の割合の極限」と位置付けるとき、まず「変化の割合」に着目し「独立変数の変量を徐々に小さくしていく」と考えたとする。このとき、次の 2 つのことに注意を払いたい：

- 主観的ではあるが、有限の量を「徐々に小さく」とることによって、極限の状態は動きとして見えてくる
- 数学の諸概念は、厳たる系列を持つものではなく、必要に応じて獲得され、問題解決において組織的にそれぞれが機能するものである

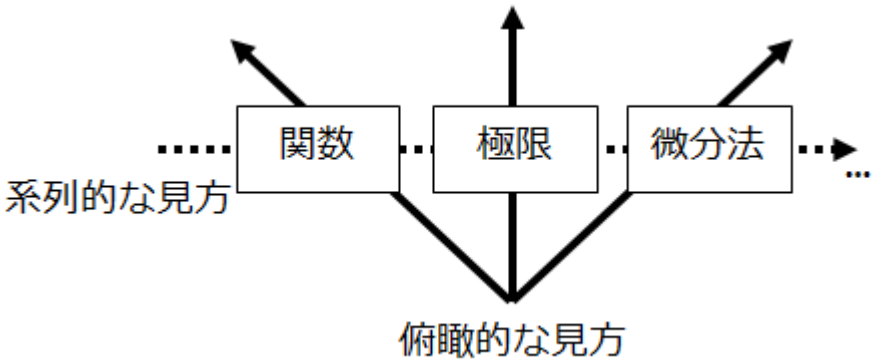


図 6 「系列的」な見方と「俯瞰的」な見方

たとえば「微分」を理解するにあたっては、有限の概念ーコンピュータの考え方の起点であるーに重きを置くことで、「極限」は「微分の前に理解すべきこと」というよりは「微分の中で理解すべきこと」であることが明らかとなる。

6. 暫定的な結論

本稿では、いくつかの視座・視点・視野によって、数理的な考え方と、算法的な考え方の違いを考察した。両者は、数学という世界の中で遠からずとも異なる考え方であることが伺える。仮説構築のための暫定的な結論として、表 1 に数理的な考え方と、算法的な考え方の違いを示す。

表 1 「数理的な考え方」と「算法的な考え方」の違い

	数理的な考え方	算法的な考え方
問題の理解	俯瞰的	系列的、または組織的
抽象化	凝集的 すべて（All）を網羅する	捨象的、効率的 入力が必須であり、そのそれぞれ（Every）に対応する出力を返す
スナップショット	ー	プログラミングにおける性質であり、思考の外化に通じる
変換の容易さ	公理からアルゴリズムを導くことは比較的容易である	アルゴリズムから公理を導くことは比較的困難である

表より、数理的な考え方と算法的な考え方の違いは、絵画と文芸作品のような違いに通じるようにも見える。算法的な考え方を表現するプログラミング（アルゴリズム）は、おそらく数理的な考え方を補完するものであり、少なくとも数学を「操る」ためには有益であろう。

今後は、仮説の証明、および算法的な考え方に基づいて、数学の諸概念の学習法を確立していくことが課題となる。

●参考文献

[1] 文部科学省「プログラミング教育実践ガイド」一般社団法人ラーン・フォー・ジャパン，2015.

[2] たとえば 小泉真也, 濱田百代, 佐藤元彦, 澁谷久, 「プログラミング的文脈が数学の理解に及ぼす影響」, 公益社団法人 日本数学教育学会 第 49 回 秋期研究大会発表集録, D7, pp.123-126, 2016.

[3] A.D.Aleksandrov, A.N.Kolmogorov, and M.A.Lavrent'ev, eds., Mathematics: Its Content, Methods and Meaning 1, Cambridge, Mass: MIT Press, 1963.

[4] ブライアン・バターワース,「なぜ数学が『得意な人』と『苦手な人』がいるのか」, 主婦の友社, 2002.

[5] 新井紀子, 「こんどこそ！わかる数学」, 岩波科学ライブラリ 128, 初版, 第 7 刷, 岩波書店, 2009.

[6] 銀林浩, 「子どもはどこでつまずくか」, 初版, 一刷, 国土社, p.184, 1994.

[7] 戸塚滝登「コンピュータ教育の銀河」, 第一刷, 晩成書房, pp.176--179, 1995.

[8] Marvin Minsky, "The Society of Mind", Chapter10 New York, NY: Simon & Schuster. pp. 98–107, 1998.

[9] マーヴィン・ミンスキー著, 安西 祐一郎訳,「心の社会」,産業図書, 1990

[10] Donald E. Knuth, Algorithmic Thinking and Mathematical Thinking, American Mathematical Monthly, March 1985, pp.170-181.

[11] 有澤誠 編「クヌース先生のプログラム論」共立出版株式会社 pp.21-43, 1991.

● 英文タイトル

Does programming education dream of mathematics teacher?

● 英文要約

Mathematics is a base of information science. Information science is a base of programming. Mathematicians, information scientists, mathematics teachers, informatics teachers and so on, all concerned have a clear understanding the relationships. However, they answer evasively about differences between mathematics and information science.

Do programming sources become expositor of mathematics? Before we get started, the authors consider differences about between the two.