長野大学紀要　第44巻第１号　　27―39頁（27-39頁）2022

無限をどう学ぶか
――プログラミング教育の曲がり角にいて――

How to Learn Infinity
――Thinking about Near Future of Programming Education――

平　岡　信　之*

HIRAOKA Nobuyuki

1　はじめに

筆者は大学教員として着任して以降、約30年に渡ってプログラミング教育に携わってきた。本稿では、筆者がその経験を通じて得た知見と、喫緊の課題に関する認識を「無限」を軸にして整理し論述していく。

1.1　変化したものと変化しないもの

プログラミングは、デジタルコンピュータがストアドプログラム方式（いわゆるフォン・ノイマン方式）を採用した時点から自動的に必要性が発生し、機械語命令（インストラクション）をデータとして（２進数などで）表記する原始的な方式からそれに対応した人間可読性の高い命令語を列記する方式（アセンブリ言語）を経て、工学的に設計されたプログラミング言語で記述する形に発展して、（原理的にはそのまま）現在に至っている。一方、情報技術（後には情報通信技術、ICT）全般を見ると、この間にいくつかの劇的な変化が起きている。デバイス技術の発展に伴うダウンサイジングと性能向上、ネットワーク技術との融合、可搬化、それらの集大成としてスマホの普及、こういった変化の影響により我々の社会や暮らしや心のあり方も変化した。当初はプログラミングという営みは専門職かオタクに限定されたものだったが、この大きな動きの余波を受けるようにして、昨今は初等中等教育から開始して誰もが取り組むべき学習課題になってきた。

コンピュータの内部ではメモリとディスクなど処理速度が何桁も違う技術が同居していてその間の調整が重要な技術（キャッシュ等）にもなっているが、現実世界でも、変化の速度の速いものと遅いものがあり、その差が顕著になってきている。プログラミングはICTの一角を占めるものではあるが、前述のようにその基本原理に大きな変化はなく（小さな変化は後述する）、ICTの中では亀グループ（兎との対比でこう呼んでおく）に入るものだと考えられる。

とは言え、（極言すれば）自然発生に任されていた選択科目時代のプログラミング教育が、必修科目時代に入ると、その目的を問われるようにもなった、これは大きな変化だと言える。ただ、目的を論じることは本稿の目的ではない（現時点では筆者の手に余る）ので、プログラミング教育の目的意識が、

技術者養成<ー>教養（思考力涵養）

の両極の間でスペクトラム分布しているらしいことだけを指摘しておく。学ぶ側の環境変化（言語処理系や技術情報のネット流通の浸透）が自然進行していたその動きを、社会的変化の方が一気に追い抜いた、というのがこの数年の動きだとも言えるが、その是非については本稿では言及しない。

1.2　言語と思考の関係

前節ではプログラミング技術の変化を亀グループ呼ばわりしたが、それでもこの60余年の間に変化は起きている。その変化の中で、パラダイムシフトとも称されるレベルの事項を（諸説あるが）３つ挙げておく。

1．構造化：GOTO文による制御の遷移を乱用することで複雑に絡み合う「スパゲティコード」を生成してしまう危険性の指摘から、構造重視のプログ

企業情報学部准教授*

ラミングスタイルを確立させた。

2. オブジェクト指向:　コードとデータの主従関係を逆転させ、データ（オブジェクト）にコードが従属する形が作られた。
3. 関数型記述、論理型記述:　動作記述ではなく宣言的な記述を行うための土台となった。

これらの新しい発想によるプログラミングを実現させるために、プログラミング言語が新たに作られ（または改良され）普及することで、プログラマー全体の発想法、設計手法もゆっくりと変化してきた。自然言語においても、言葉とそれを使う人々のものの考え方が相互作用されると言われるが（例えば [Ted1]）、プログラミングにおいても、思考の土台となる言語（以下、本稿では単に「言語」と表記したものはプログラミング言語を表す）があり、言語は思考に密に影響を与えると考えられる。

なお、上記の変化の周辺ではマルチプロセッサ、ネットワークといったハードゥエア環境の進展も進行しておりその影響も受けつつあるが、その影響は現時点では定まったプログラミング言語の形に昇華しておらずここでは大きくは扱わない（ただし後で少し言及はしたい）。

1.3　現場の状況

筆者が大学でプログラミング入門を扱ってきた30年弱[1)]の期間を、主に使用した言語に基づいて、以下のように 3 期に分けることができる。それぞれの時期における選択には教室の情報環境の制約と発達も関係している。

1. C と Logo:（Unix端末としても利用できる）DOS パソコンの時代。無償で使える言語処理系としてUnixに添付のCコンパイラと国産PCに添付の Basic しか見当たらなかった時代でもある[2)]。
2. Visual　Basic（VB): Windowsの時代。ツールの選択の影響もあり、GUIアプリケーションを作ることが中心テーマになり、イベント駆動のプログラミングの比率が高くなった。
3. スクリプト言語群:ネットの普及の結果、様々な言語の処理系が（スクリプト言語を中心に）無償で流通し、言語の選択肢が急速に広がった時代。

言語を適切に選ぶことの重要性を強く感じつつ、窮屈な（選択肢の乏しい）時代を通過して、潜在的には自由な時代（上記の第 3 期）に入ってからも、実はさほどの冒険はできずに現在に至っている。その理由として以下の要因があった。

- 学びのインセンティブとしてのアミューズメント要素: LogoやVBは、お絵描きができる、という点を重視して選択したもの。スクリプト系言語は概して初学者の最初のハードルは低めに作られているが、グラフィック機能が標準装備されている処理系は殆どないので、第 3 期に入ってもこの要素が言語選択の大きなポイントになった。
- 情報源の豊かさ : 日本語で書かれた平易な教科書がないと、教科用言語として採用するのは困難になる。
- 様々な同調圧力 :
  - 第 1 期の頃には COBOL を使うべしという強い圧力と戦っていた。産業界でのシェアや卒業生の就職先（の候補となる企業）での利用実績がその圧力の根拠だったようだ。
  - 資格試験（特に情報処理技術者試験）の合格実績を上げようという動きも発生する。そのため、資格試験での採用言語を使うべし、という圧力も強くなる。
  - 第 3 期になると、Javaが産業界でのシェアを拡大させていた（昨今はPythonがそれに代ろうとしているようだ）。
  - 学部の中でも学生に提供する言語を 1 つもしくは少数に纏めようという動きがある。その際には「この科目でこのライブラリを使うためにこの言語の入門教育が必要」といった要求も合わせて調整する必要がある。

ここで名が挙がったCobolやJavaは、ボイラープレートの多い言語とされている。こういった言語は（その言語の記述力などの特性をさて置いても）以下のような問題がある。

- 初学者の学習を阻害しやすい（ボイラープレートを覚えることでプログラミングが解ったような気分になってしまう）。

1) 最近の筆者は入門教育の枠では、プログラムを作るプログラミングを用いない情報学入門を行っており、本稿で対象としているプログラムを作るプログラミング言語の範囲からは外れているが、それについては改めて別稿で述べることとしたい。

2) 正確に言えばUnixやPCが有償なので、無償というよりは「おまけ」が相応しい。

• 別の言語に移る際の抵抗にもなる（覚えたものを捨て去ることの抵抗感）。という理由で教える立場の者は（思考力養成を目指しているなら）なるべき忌避したいものだが、時にはそういった要素も加味しつつの言語の選択も必要になってくる[3]。

本節では筆者の経験を語ったが、おそらくどの現場でもこういった葛藤の中で妥協をしながらの教育活動が行われてきた歴史を持っているだろうと推察する。

局所的な悩みどころの例

例えば変数 a と変数 b の値を交換したいという場面、典型例としては互除法で 2 数の公約数を求める手順の前段階で出会う、大きい方を a に置きたい時、これをどう解説するか、いくつかのトピックと選択肢がある。

1. 狭い道路や単線の線路での入れ換えのイメージを語る（伝統的な手法）

```
var tmp=a ; a=b ; b=tmp
// のようなコードを示す
```

2. 多くの言語で関数呼び出しは基底型の引数については call by value であり

```
function swap(x,y)
{var tmp=x ; x=y ; y=tmp}
// を用意して
swap(a,b)
```

は機能しないことを伝える。

3. マクロ（多分に言語に依存するが）なら機能することを伝える。

```
#define swap(X,Y) int tmp=X ;
X=Y ; Y=tmp;
// を用意して（整数型限定だが）
swap(a,b)
```

4. 多重代入（並行代入）ですませる。

```
a,b = b,a
```

これがあれば 1 .〜 3 .は過去の遺物であり知る必要もないとも考えられる。

5. そもそも mutableな変数の濫用を戒めて、交換をさせない。

```
a>b ? gojo(a,b) : gojo(b,a)
```

のような代案を示す。

これらを逐一伝達しようとすると時間は足りないし受け取る側も消化に苦しむだろう。どのようなレベルとポリシーに基づいてトピックを取捨選択するか、プログラミング技術がそれなりに歴史を重ねてきて今なお進歩の途上である現在、教える側にも悩みどころは満載である。本稿ではその悩みどころを提供することで広く議論の材料としたい。

2　ゼロと、無限に大きな数

伝統的な数値表現

「無限」に関係してプログラミングで出会う概念の 1 つは、数値の大きさとしての「無限大」と「無限小」だろう。そもそもコンピュータは有限寸法の電子回路で処理を行っており、現実的に妥当な応答時間で結果を返すことも求められるので本当の意味で無限に大きな数値を扱うことはできない。（無限小の方、すなわち精度が限りなく高い数値については、初級の段階で出会う場面はなさそうなのでここでは省略する）。コンピュータのハードウェアで現実的に行っているのは、そのハードウェアで定められた 1 ワード（典型的には 32bitや64bit）を単位として、整数型表現と浮動小数点型表現に分けることだった。この分け方自体は、連続量と離散値に別れた数学の体系にも合致しているので問題はない[4]。ただ、整数の許容ビット数を超える大きな値が生成されると、オーバーフロー例外が発生するか、言語によっては勝手に浮動小数点数に変換が行われる。

```
// JS
Welcome to Node.js v17.8.0.
> 2**69
590295810358705700000
> 2**70
1.1805916207174113e+21
```

プログラマーは、整数の（場合によっては対象マシンによって変化する）有効ビット数を意識した上でコーディングを行うのが従来の常識であった。

昨今の拡張

複数のワードを連結して 1 つの数値を保持するデータ型を作り、その型のための処理（四則演算など）を用意して、整数の桁数の制限を解除させることはソフトウェア的には可能であり、言語仕様の拡張

3) 拙稿[Hiraoka1]にもその経験を書いたので参照されたい。

4) むしろ整数と実数の間の融通を効かせすぎる言語が時に予想外の挙動を示すことがある。

またはライブラリでの提供により多くの言語で行われている（Scala の BigInt、Java の BigInteger、Ruby2の BigNum、Python2のbignum など、型宣言や型変換を行って使うもの）。さらに、最近は、桁数制限のない整数型を従来からの基底型としての整数型とシームレスに融合させた言語も増えつつある（Ruby3, Python3, Haskellなど）。整数に限って言えば、ハードウェア実装上の都合をプログラマが考慮する必要がない（従って学ぶ必要もない）時代に入ったと言える。

```
# Python3
Python 3.9.10 (main, Jan 15 2022, 11:40:5
>>> 2**200
160693804425899027554196209234116260252220
```

上記の、計算の結果として発生してしまう限りなく大きな数値とは別に、予め用意すべき大きな（または小さな）数があることにも言及しておく。典型例としては、数値列の中で最大のものを求める簡便なアルゴリズム（または fold, reduce のようなメソッド）で、「現時点での最大値」を保持するチャンピオン席に相当する変数に初期値として与える値や、繰り返しの停止を保証べくセンティネル（番兵）として設定しておく値を考える時、対象となるデータが自然数ならば -1 を使い、正負の値をとり得る場合にはデータの値域を予測してそれよりも小さい値（絶対に勝たない値）を見積もっておこう、というようにこれまでは唐突にヒューリスティックスが要求される場面があったが、この場面で利用できる特別な数値が言語仕様に用意される例が増えつつある。多くの場合、”infinity” やそれを略記した ”inf” というような名前がついている。その用途は上記の初期値や番兵には限られず、様々な場面で従来のプログラミングの窮屈さを緩和するべく配慮されている。いくつかの例を示す。

- 比較対象

```
# Python3
Python 3.9.10 (main, Jan 15 2022, 11:4
>>> float('inf')>2*200
True
```

- 終端を定めない範囲

```
# Ruby
irb(main):001:0> 1..Float::INFINITY
=> 1..Infinity
#なお、Ruby では以下の書き方もある
irb(main):002:0> 1..nil
=> 1..
```

- 0 除算の結果としての値

```
// Scala
Welcome to Scala 2.13.8 (OpenJDK 64-Bit S
scala> 1.0 / 0
val res1: Double = Infinity
```

これらは現時点ではそれぞれの言語に固有の工夫・拡張でありプログラミング界に遍く定着したものではないが、今後の動向はここから読み取ることができる。こうして見ると、古いプログラマーが金科玉条のように語っていた「常識」（決してゼロで除算をするな、事前に場合分けをしろ、など）が崩れて行きつつあることも感じとれる。

3　無限のデータ空間

3.1　配列データ

典型的な配列データは以下のような書き方で扱われていた。

```
// C
#define MAX_SZ 1000
int buffer[MAX_SZ];
for(i=0 ; i<MAX_SZ ; i++) {
    // do something
}
```

C 言語の時代までは、データと制御情報（典型的にはその寸法やデータ数）を一括管理する表現方法が用意されていなかった（文字列でさえその長さを表す情報を直接には持たず終端記号を検索することで情報を得ていた）。また、ひとかたまりの配列データはメモリ空間の連続した領域に配置される必要があり、そのためにプログラマは事前に寸法を定めてその領域を確保する必要があった。この制約によりハードウェアの機能（インデクシング）を活用して配列要素に 1 インストラクションでアクセスでき、また高速化のための機能（CPU内のパイプライン等）も有効に働くという性能上のメリットがあるが、それと引き換えに配列寸法についての自由度がなく、プログラマは予め余裕を見てデータ量を見積もっておくなどの（ここでもヒューリスティックな）工夫を要求されていた。

一方で、早い時期から柔軟なデータ構造を目指す言語も作られてはきている。その典型例がLispであり、ここで主役となっているデータ構造である連結リスト（linked list）は、自由度を体現する（配列に対する）もう 1 つのデータ集合体として認知されることになった。連結リストにおいては、長さ（要素数）の増減は事実上無制限である（ 1 次元リスト以外の応用範囲もある

が本稿では省略する)。後に出現した言語では配列と連結リストの両立、若くはいいとこ取りをする形で伸縮自在な配列データ構造が提供されるようになっている。特に、迅速な開発が求められる時代に重要性が高まってきたスクリプト系言語では、(ハードウェア資源等の制約による現実的な限度はあるものの)寸法に仕様上の制限が設けられていないデータ構造(配列やリスト)は重要なポイントになっている。もちろん使う側もそれを活用する技が求められる。

3.2　整数列の扱い

ハードウェアの条件ジャンプ命令から出発(後に繰り返しの構文に発達)したプログラミング界では数値列を扱う際にループ(繰り返し)で実現する考え方・書き方(例えば前節に示したC言語のコードのように)が広く定着しているようだ。逐次的に順々にデータを取り出す、言葉を変えれば「時間軸上にデータを並べる」考え方である。

それに対して、データ列から出発するプログラミングの発想法もある。その考え方を最初に言語として実現したAPLでは `⍳ n` という式(最初の文字はギリシャ文字のイオタ[5])で1からnまでの自然数列を生成した。現代の言語では実体としての整数列を保持した配列やリスト、だけでなく、開始値と終値を指定して順々に整数列を生成できる、言わばデータの種となるもの(Range型などと呼ばれることが多い)も好んで使われる。このデータ列についても、昨今は「無限」の発想が入り込んでいることにここで言及しておく。前章で例示したRubyでの `1..nil` や、Haskellの`[1..]`がそれに相当する(その使い方は後の章で示す)。

3.3　表計算での発展

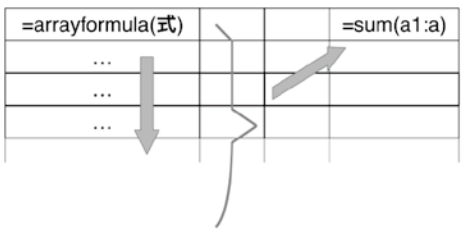

図1 制限なしの範囲

図1

同様の制限緩和はスプレッドシート(表計算)でも静かに進行しており、表計算の使い方も変化しつつある。その主眼点は2つ。

1. 列(または列範囲)全体の指定(下端を定めないセル範囲指定)
2. 単一の数式を複数セルに適用させる機構これまでの表計算の常識では、ある範囲の各セルに計算式を表示させるためには、該当する各セルに数式を入力(一般にコピペで複製する)する必要があった。数式の対象となる範囲についても、上端と下端(左右についても同様に)を参照式(またはドラッグ操作など)で指定た有限範囲でなければならなかった。その制限を撤廃する式が主要表計算ソフトには用意されていて(例えばGoogle SpreadSheetでは `a:a` または `a1:b` のような範囲参照と、arrayformula関数; 第1図を参照)、表計算ユーザが数式を該当範囲に大量生産で複製するなどの手間を要しない時代に入ったと考えられる(ただしこの機能拡張はソフトウェアもしくはWebサイトによって個別の記法が使われていて、現状では統一したルールにはなっていない)。

表計算は教育の仕方によっては「神エクセル」という言葉で忌避される見せかけを取り繕うためのツールに成り下がる危険のある道具だが、Reactiveプログラミングの理解を導くための好適な題材でもあり適切な場面で教程に組み込んで活用していきたいものである。

4　関数的な思考とコーディング

4.1　関数型プログラミング前夜

関数と手続き

本稿で既に何度か言及しているC言語は、現在もなおプログラミング界で重要な地位を占めており、後の言語に(良くも悪くも)強い影響を与えたことは間違いない。その影響の1つが、「関数とサブルーチン手続きが同じ顔」である。それまでの言語では両者は区別されていたが、C言語ではプログラムのひとかたまりを、その主目的が作用なのか値なのかを問わず、統一的にfunction(関数)と呼んだ。元来(数学的な)関数とは与えたもの(入力)と得られるもの(出力)の関係を意味していて、「状態変化」の概念は含まれないが、手続き型言語を関数の用語で既定した結果として、C言語は関数の中に「副作用」を持ち込んだことになった。ひとたび混ざってしまったものは分離させるのは困難で、この状態は今も(現場でも教育の場でも)継続して

5) iotaの名前は後の幾つかの言語にも継承されている。

いるが、それを改めて分離しようという動きはゆっくりとであるが進行している。

スカラーからベクターへ

C 言語の頃には、関数の引数、値、変数の値などの主要データは 1 ワードのものに限定され、標準提供される演算子もそれに準じていた（それを超える寸法のデータ群はポインタで受け渡しが行われた）。その結果、プログラミングはスカラー単位で行うという考え方が定着した。

複数ワードにまたがるデータのかたまりを一括で処理するという発想は、実は APLやLispの頃から存在していた（ハードウェアが未成熟という当時の事情もありマイナー言語の域を出られなかったが）。これも、Perlから始まるスクリプト言語の時代になって初めて広く知られるようになった。

4.2　プログラミングの発想の変化

例えば、1 から n までの数列についてそれぞれを 4 乗した値の列を求めたいとき、古い言語では、最初に結果を格納するための寸法 n の配列を用意し、ループを回して 1 つずつ計算させていた。昨今は以下のように高階関数を使う書き方が一般化してきた[6]。

```
# Ruby
(1..n).map{|x|x**4}
```

その中から数字'6'を含むものを選び出すなら、

```
# Ruby
(1..n).map{|x|x**4}.
select{|x|x.to_s['6']}
```

のように、メソッドチェーン（Rubyの場合）で実現することができる。

第2図

図2

なお、Ruby（オブジェクト指向言語の 1 つ）でのメソッド呼び出しは、オブジェクトを 1 つメソッドに渡して（メソッド内では self として参照する）、必要に応じて付加的に引数も渡した上で、値（オブジェクト）を 1 つ返す。オブジェクトとメソッドの関連付けにおいてオブジェクト指向独自の仕組みは内在しているが、非破壊的なメソッド呼び出しを行って値を返させ、されにその値を別のメソッドに渡すという連鎖構造（第 2 図）の仕組みは、関数型言語のそれと意味的には同じであり、関数型プログラミングの枠組を提供するどの言語を使っても、同様の構成で提供されている部品を組み合わせて、同じ発想法で作譜が可能である。

5　無限データ列

5.1　「無限」が忌避される背景

プログラミングを学ぶ場面では、「無限」という言葉はともすれば毛嫌いされがちである。そこには「無限ループ」のネガティブなイメージがある。プログラミングを語る（我々教員を含めて）古いプログラマーの中に残る、CPU時間に課金されていた（しかもバッチ処理の）時代の記憶、が根強く残っていることも一つの要因だろう。実際、その時代はループを記述することによるコーディングが主流だったこともあり、無限ループ発生のリスクは高かったし、その際のダメージ（巨額の利用料請求など）も大きかった。もしかすると「無間地獄」という恐ろしげな言葉の響きとの共通性も関係するかも知れない。

しかし、現在は、データ列を出発点とするコードへのシフトも進み、TSSの出現以降はインタラクティブな開発が日常のものになり、ループすれば（または予想以上に長時間になれば）割込操作でプログラムの停止が可能になっており、すなわち「無限」を不必要に恐れる必要はなくなっている。むしろ今後の学習においては、「無限」にも目を向け、「制御可能な無限」「制御不能な無限」に分けて認識し、後者だけを警戒する、というスタンスが望ましいだろう。

5.2　無限データ列の扱い方

前々章で終端を指定しない整数列を紹介（だけ）した。これが無限に続くデータ列の代表例だが、これを（前章のようにデータを出発点とした）プログラムに生かすためには 2 つの条件が必要になる。

1．（処理系の要件として）遅延評価、またはそれに準ずる機能を持つ
2．（使い方として）後ろの段で適度な長さにカットして出力させる前者の条件を満たす代表的な言語としてここではHaskellを使って例示する。ここでは無限リストが登場するが、無限リストが生成されたり関数に引数として渡されたりする時に、値の評価はリストの先頭から順に必要なだけ（大

6）言語によっては内包表記で書く方がスッキリするので好んで使われるが、mapやselectメソッドを組み合わせた表記と内包表記は意味的には同じなので本稿では後者への言及は行わないことにしておく。

抵は出力の段階で必要とされるだけ）行われるため、円滑に応答は返ってくる。

```
-- Haskell
Configuring GHCi with the following
packages: ghci> take 10 $ map (^4)
[1..]
[1,16,81,256,625,1296,2401,4096,
6561,10000]
```

これは前章（Rubyで書いた）に準じて自然数の 4 乗の数列を無限リストとして生成（$の右側）し、その中から先頭10要素だけを選び出すフィルター関数（$の左）に渡している式[7]。後者（左側）で上記 2. の条件を実現している。

5.3　無限データ列の積極的活用

ここでは無限リストを積極的に活用するプログラムの例を 2 つ提示する。いずれも無限数列を生成する関数なので前節に倣って関数 take を使って個数制限をしてから出力させるか、（REPL上ならば）無限に続く出力を割り込み操作で強制停止するという前提で使う必要がある。

素数列

```
-- Haskell
erato :: Integral a => [a] -> [a]
erato (x:xs) = x: erato
[y|y<-xs, y `mod` x /= 0]

primes :: Intagral a => [a]
primes = erato [2..]
```

エラトステネスの篩を再帰呼び出しで実現している。

- 再帰の各段で関数eratoは、素数の候補となる数のリスト（昇順）を引数として受け取り、
  - その先頭の値 x を値として（関数が返すべきリストの先頭要素として）生成し、
  - リストの残りの部分（tail部、つまり cons演算子 ':' の右側）の生成は次段の再帰呼出に委ねる。
- 次段に渡す引数（ここでは内包表記で書いてあるリスト）は、自段が受け取ったリストの中から x の倍数を取り除いたリストを意味しており、
  - この時点で x とそれよりも小さい（即ちここまでの段で生成済の）素数でのフィルタリング（割り切れないことを確認）は完了していることになり、そのリストの先頭要素が素数であることもわかっている。

フィボナッチ数列

実は n 番目のフィボナッチ数を求める式の再帰的定義として有名なものは以下のような式であるが、

```
-- Haskell
fib :: (Eq a, Num a, Num p) => a -> p
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

この定義だと二重再帰であることに加えてそれぞれの値を求める毎に枝分かれしつつそれぞれの枝について 1 に到達するまでの深さで再帰呼出が発生するため、関数のメモ化が行われない限り現実的な応答時間での計算ができない。同じ二重再帰でも下の式のように上昇方向の再帰だと現実的な動作をする。

```
-- Haskell
fibs :: Num a => [a]
fibs = 0:1: zipWith (+)
fibs (tail fibs)
```

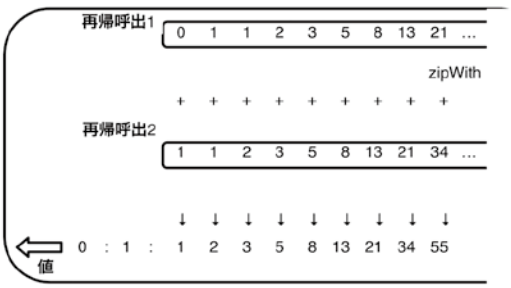

図3 関数 fib の構造

図3

ただし、このコードは、再帰呼出の各フレーム相互の間の関係をイメージする必要があり動作を理解するための難易度は高いかも知れない（第 3 図）。手続き型言語の枠組の中で暮らしていると、変数の値（などの、実行状態）がプログラムの進行につれてどう変化するかを時間軸上でイメージすることが重要になるが、本稿で重点的に提示しているコーディングスタイルでは、

7) 関数型言語としての機能を高度化させた言語では、高階関数（ここではmap）に引数として渡す無名関数（ここでは 4 乗させる関数）をラムダ式ではなくカリー化した部分関数として渡せるので前章の例よりもコンパクトな記述になることにも着目されたい。

データ列の中でデータがどう流れて行くか、再帰の各フレーム相互間でどうやりとりされるか、といったデータの流れ（データフロー）を想像することが重要になってくる。

5.4　富豪についての再考

例えば 1 から n までの数値を扱う時に、この一連の数値を（1 つの変数で）時間軸上に配置するか n 個分のメモリ空間を使って空間上に配置するか、これまでの議論はその選択を論じたものでもあった。ところで、ふた昔も前だが、富豪的プログラミングという言葉が一部で話題になった時期がある[Masui1]。プログラマーがともすればコンピュータ資源（メモリ領域など）を節約しようとしてしまう傾向に対する批判的な言論だった。この言葉が人の口に上らなくなったのはこの考え方がある程度人心に定着したからだと考えられるが、その割に今なお空間を節約したがる人も多いようだ。

本章の終わりに、これまで論じたよりもさらに富豪的な発想で行われているプログラミングスタイルについて言及しておく[8]。

```
⍝ APL
n←30
⍞←(~R∈R∘.×R)/R←1↓⍳n
```

- APL言語では右から左へ向けてプログラムが実行されていく（と理解していい）。
- 前々章で紹介した `⍳n`（ただし `1↓` で最初の要素を取り除くので、2 から n までの数列もしくはベクター）を変数 R に一旦代入し、
- `R∘.×R` でその外積を生成する（R に含まれる任意の 2 数の積を全て含むサイズ(n-1)×(n-1)の行列になる）。
- R の各要素が上記の行列に含まれるかどうかを（演算子 `∘` で）検査し
- その結果として、論理値からなるベクター
  `0 0 1 0 1 0 1 1 ...`　が得られるが、
- その（各要素の）否定（演算子 `~`）をとったベクター（の各要素）を右辺として
- select 操作に相当する演算子 / を R に適用することで、R の中から行列に含まれない要素（つまり素数）を選び出す。

式が短い割に言葉にすると長いが、APLではこんな発想でプログラムが作られる。このAPL言語が作られた1960年前後はハードウェア能力を初めとした諸状況がいずれも不十分だったために世間に広がらず今に至っているが、こうしたアイデアの中にも改めて発掘すべきものが眠っているのかも知れない。

6　並行動作のイメージ

6.1　動作条件

前章では無限列を扱うコードをHaskellで示したが、この考え方によるコードの動作は、必ずしもHaskellのような遅延評価に基づく言語である必要はなく、選択的に遅延評価を指示できたり、それに必要なデータ構造と操作が提供されていればいい[9]。このコードを実現可能な枠組みが提供されている言語を（筆者が調べた範囲内で）類型化して示しておく。

1. ストリーム（遅延評価できるリスト）と、それを対象とした cons操作:
   - Scala : scala.collection.immutable.Stream と **#::** 演算子
   - Coconut : (| ... |) (banana brackets)による lazy iterator と、:: (iterator chaining)演算子
   - Scheme : util.stream と stream-cons 関数
2. ジェネレータ関数（またはコルーチンとも呼ばれる）
   - Ruby : Enumerator によるジェネレータ生成と、Enumarator#lazy, Enumerable#lazy
   - Elixir : Stream.unfold 等によるジェネレータ生成（ただし、この unfold による遅延された繰り返しは、一本の列に限定されるため、前節の fibs関数のように多重再帰で実現できるコードはこのままでは扱えない）

6.2　並行動作を記述する仕組み

前節では、前章に示した例題に適用可能な言語を挙げたが、正格評価を原則とした言語でも並行動作を実現する技術は多数開発されている。その代表的なものはマルチスレッド技術だが、ここでは前々章の無限数列に関係する技術を整理しておく。各項の末尾で無限自然数列を生成する例も示しておく。

8）このコードは[Web1]を参考にした。

9）遅延評価を選択させるための余分な記述は最低限必要になるためいずれも Haskellに比べるとやや長めのコードになるが、ここでは気にしないことにする。

1. Stream:前節で挙げた以外に、有名なところでは Java（ 8 以降）で Stream が提供されている。Stream の機能自体はどの言語でも原理的に同じだが、Stream を生成したり操作したりするための関数やメソッドのラインナップはまちまちであり、その違いによって応用可能性や書きやすさに差異がある。なお、C 言語やそれを直接継承する言語（Pythonも含めて）では、バッファリングされた入出力（のいずれか）を行うための抽象データ構造をストリームと呼んでいて、目的が違うものが同じ名前を持つことで、言語界全体では混乱の 1 つになっている。

```
# Elixir
Stream.iterate(0,&(&1+1))
|> Enum.take(30)
```

2. ジェネレータ関数:通常の関数は一回の呼び出しに対して値を 1 つ返して終了する。ジェネレータ関数は、状態を保持し、値を複数回返すことができるもの。Pythonでは return文 の代わりに yield文を使うことで記述し、JavaScript では function の代わりにアスタリスク付きのキーワード `function*` とyield文で定義する。

```
# python
def nat():
  n=0
  while True:
    n+=1
    yield n
import islice from itertools
 # iterator  にスライス構文を直接使えないので
list(islice(nat(),30))
```

3. パイプライン（Unix系OSの技術）:流れ作業の上流と下流のプログラムが並行して進行する動作モデルは、OSの機能として実現されているものをシェルから平易な記述（パイプ記号）で呼び出す記法を創造したUnixオペレーティングシステムがルーツになっている。初学者はプログラミング言語で試すよりも前に、Unix系シェルで以下のような動作を経験しておくと理解しやすいだろう。

```
# Unix 系シェル言語
yes "" | nl -ba | head -30
```

なお、上記のパイプ記号は、第一段階でプログラミング言語の並行動作モデルに影響を与えた後、第二段階として、パイプ演算子（一般に記号 |> が使われていて、本稿でも作譜例の中でいくつか登場している）の形で影響を与えつつある。左から右へと読み進められるスムーズなプログラムの進行は、オブジェクト指向のメソッド呼び出しの構文（一般に レシーバー . セレクター の形）を多用する言語で先行して実現しつつあるが、この構文を持たない関数呼び出し主体の言語でも、パイプ演算子で接続することで、左から右へと流れるコードが書きやすくなる。

6.3　プログラムの骨組の将来像

プログラムは何らかの（制御された）ループで作られる。その主要な骨組のパターンは時代の進行につれて概ね以下のように変化してきたと考えられる。

1. 入力されるデータに応じた自律的ループ : 従来から（構造化以降）使われている繰り返しの構文で制御される単一プロセスのプログラム。
2. （ユーザの操作に応じた）イベント駆動のループ : インタラクティブなプログラム（特にGUIの枠組の中で動作するもの）はこのパターンが多い。
3. 複数のイベント源から発生するイベントを処理するループ : 典型的にはネットワークプログラム。GUIからのユーザの操作と、ネットワークからの情報受信とが非同期に発生する。

3．については、Unixのselectシステムコールに見られるような、複数のソースからのイベントを一括で待機する単一のループで回しつつ、受け取ったイベントの種類に応じてディスパッチ（場合分け）を行うスタイルが現在も広く使われている。しかしこの骨組でのコーディングは複雑になるのを防げない。例えばユーザ側のスレッドとネットワーク側のスレッドとで協調しながらの並行処理、のような新たな骨組みモデルを確立していくことがこれから必要になってくるだろう。

7　思考の道具としてプログラミングを鳥瞰する

ここまでの章で技術の変化に伴ってプログラミングがどう変わって来たか（もしくは、変わろうとしているか）いくつかの観点から述べてきたが、最後にそのまとめとして思考の道具としてのプログラミングの変化についても論じておく。

7.1　視点の複数化

一般に逐次型プログラミングでは、コードと実行フレーム（CPUの制御状態）が 1 対 1 に対応している（第４図 a ）。作業手順書とメモ用スペースを兼ねた紙をイメージして、サブルーチン呼び出しでは現在作業中の紙の上に一時的にサブルーチン用の紙を置いて

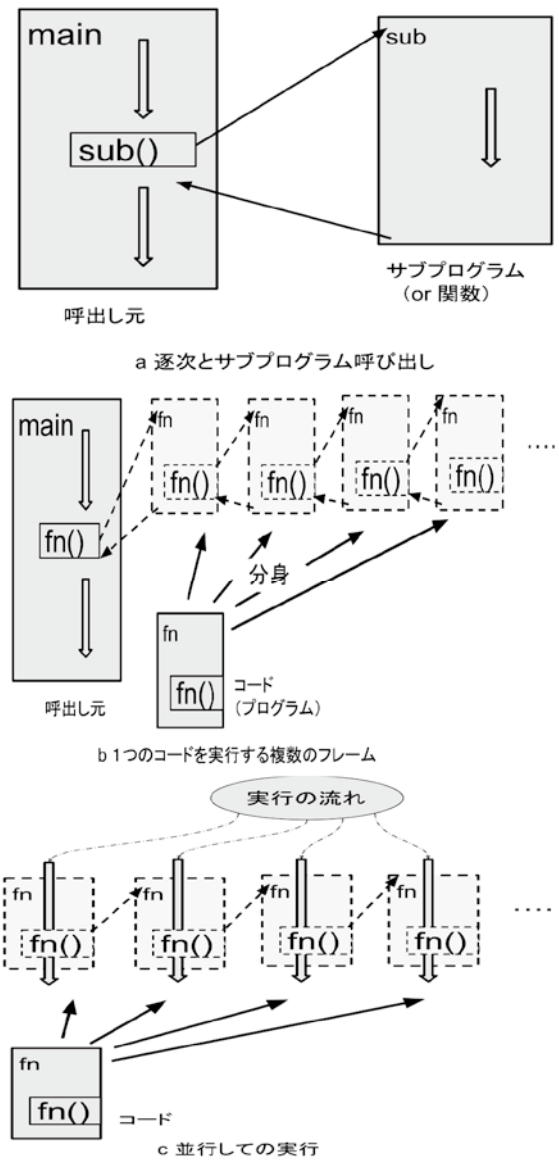

図4

（下の紙は一時的に隠蔽し）、終わったら上の紙をどける、というようなイメージで（典型的には）語られる場面である。

この対応関係に慣れた（逐次型プログラミング）の中級者に、再帰の手ほどきをしようとすると（互いに）苦労することが多いのは、再帰は、コード 1 つに対して実行フレームが複数生成される仕組み（第 4 図 b ）なので、これまで自分の頭に確立していた 1 対 1 のモデルを解体する必要がある、そのことが大きな要因であろう[10]。さらに本稿で例示した、遅延評価やコルーチンを使った再帰の場合は、生成された複数の実行フレーム（スレッドと呼んでも語弊はない）が同時並行で動作することになり（第 4 図 c ）、舞台（比喩として）1 つに実行フレーム 1 つというこれまでに自分が獲得した対応関係をここでまた破壊する必要が生まれる。実に厄介な話ではあるが、見方を変えると、プログラミングとはこのように過去の思考をリセットすることで柔軟な思考力を涵養する道場でもあると言える。

7. 2　型空間の広がり

小学校で習う算数では扱う数値に単位がついていることが多いが、数学になると抽象化が進行して単位を考慮しない一般化した「数」を扱うようになる。これは 1 つの進歩ではあるが、それでも現実の場面では数値を単位つきで考えることは重要である。初歩的な例として、例えば、速度×時間→距離 のような関係式を習って、変換の計算をする時に何が分母で何が分子だったかで悩むのはありがちなパターンだが、km/h × h → km のように単位の式にするとすんなり納得できる。

プログラミングでその単位に相当するものが「型」だと考えられる。言語処理系による型検査で引っかかってプログラムの実行までが長い道程になってしまう初心者は多いが、型を考慮することで設計がやりやすくなると共にコードの安心度が増す利点がある。現在広く普及しているスクリプト言語の多くは動的型付言語で、この30年ほどは積極的に型情報を使わないプログラミングが幅を利かせていたが、最近はそれに対する揺り戻しのように、型システムの採用も進みつつある。

静的型付言語においては、C 言語の時代の、いわば型に名前をつけるだけの素朴な型指定から、オブジェクト指向で普及した木構造で型を分類する時代を経て、言語仕様に型変数を組み込んで型の計算ができる型システムが広がりつつある。数学や物理で数値の計算に付随して単位の計算をして計算式の正当性を担保する、それと同様の型の計算ができるようになってきた。プログラミングは、値の関係をプラグラムすることと、それに付随して型の関係をプラグラムすること、と言えるようになってきており、この型システムを活用してプログラマが安心して型空間を広げていける、そ

10) immutableな変数やpureな関数についても同様に抵抗があるがここでは省略する。

のための道具立てが揃いつつある。

7.3　複雑化への対応

そもそもコンピューティングは無限長テープ上でデータを操作して単純なルールセットから多様な機能を産み出す思考実験[11]から始まったものであるが、その後、プログラミングが実用上の道具として使われるようになる過程で、多様なものを生成しようという初期の志は尊重しつつも、その産み出すものの複雑さが人の能力を超えて発散して行くことの危険性も認識されるようになった。その結果、現在のプログラミングは「抑制された複雑化」を目指すものになった。

最初の変革の契機はダイクストラの指摘[Dijkstra1]で、この時期には、GOTO文による分岐が、スパゲティ（混沌）を産み出す“harmful”なものと見なされ、「構造化」（1.2節でも触れた）が始まった。現在、構造化（に基づく書き方）はプログラマの共通認識として定着しているが、これで抑制のための作業が終わった訳ではない。mutableな変数と、副作用のある（純粋でない）関数、この2つも、GOTOとは別の形でスパゲティを産み出す可能性のあるものとして警戒視されており、その抑制策として新たな変革が動き出そうとしている。とは言え、「構造化」の浸透には時間がかかった[12]。それと同様に、次の変革もまた時間をかけて進行していくのだろう。

8　教育における課題

8.1「デジタルネイティブ」の時代

ICTに関連して子供達を取り巻く昨今の顕著な変化を要約するなら以下の2点になるだろう。

1.（日常生活）スマホ文化の普及
2.（学校で）プログラミングを含む情報教育の早期化

こうした変化は当然の結果としてその後の若者の行動様式・思考様式にも影響を与える。大学生に接してきた中での筆者の観察では以下のような事項が目立つ。

- やりたい事は「メニューから選択」
- コミュニケーションはアイコンかスタンプかスクショ

情報技術に初めて接する大学生が殆どだった30年前と比べると、ICT経験者が多い今の大学生は、ある意味で頼もしい存在ではあるが、仔細に見ると、（デジタル技術で実現された）アナログインタフェースに慣れているだけの者も多く、早くから学んで既に慣れ親しんだ知識を上書きするような新たな知識の吸収に手こずる（マスキング効果なのかインプリント現象なのか不明だが）者も多い。初学者が最初に使うべきプログラミング言語は、日本語（を典型例とする自らの母語である自然言語）である、というような議論を（改めて学び直してほしいという願いも込めて）よく見かけるのもそれと関係しているかも知れない。

8.2　若者の「世界観」

若者の世界観に関連しての筆者の心配事を、本稿で扱ってきたプログラミングのパラダイムとの関連で2点だけ述べておく。

気づかずにプログラミングされている自分

30年前と比べて、ICTに関連する教科書や入門書は格段に親切になった。特にアプリケーションのガイド本等では、図版が豊かに提示されていて、その指示通りに進めて行けば着実に、学習（もしくは課題の遂行）が進められるように作られている。（が、見方を変えれば）この時この教科書は、学習者をターゲットマシンとしたプログラムであり、学習者は知らないうちにプログラム実行マシンと化していることになる[13]。これに慣れた若者は大学の授業でもこの親切なガイドを求めるようになり、授業を行う側も（頑張らないと授業評価の点数が下がるので）その圧力に抗えず不本意でも親切な逐次型ガイドを作ってしまう。こうした需要側供給側の微妙な駆引の蓄積の結果、道案内、製品取説、さまざまな領域に逐次型の指令書が蔓延しているようだ。

プログラミングは、命令的に表現する考え方と、宣言的に表現する考え方がある。その前者を偏重して採用してきたこの社会の危うさを感じており、多様なパラダイムを織り交ぜて社会に伝えて行かねばと考えている。なお若者たちは「明日、晴れれば駅前広場で待ち合わせて、雨ならコンコースにしよう」のような日常生活でのプログラミング思考の機会は徹底して剥奪されている。

11) いわゆるチューリングマシン[Turing1]。

12) それでもなお、例えばC言語にgoto文は現存していて、ということはまだ使う人もいるということだろう。

13) この指摘は情報分野には限らず、例えば、数学などの「解法」を覚えて適用する学習など、知らないうちに広く蔓延している根深い現象ではあるが、おそらく逐次プログラミング的思考の（教える側への）浸透もそれを後押ししているだろう。

箱庭的世界認識

長く生きてきた身として、世界は果てなく広いし、知識は無限だ、というようなことを若者に伝えたい、筆者はそう考えて教育の仕事に携わっている。ところがプログラミングを扱う場面になると、ここまで述べてきたように実装上の制約や伝統的情報教育課程の制約により、対象空間（データ列など）を有限の枠内に限定して、そこからはみ出さないよう「注意深く」振る舞うことを第一義とするような教程を作らざるを得ない、それがこれまでの教室像だった。

せっかくプログラミングという広大な可能性を持つ技術もしくは趣味を紹介していながら、いわば、箱庭で遊ばせるような教材の提供をしてきたことになる。プログラミング教育が、（精神的に）「広い運動場が与えられてるのに四畳半のスペースでしか遊べない子」を作り出してしまっていないかという懸念は、既にある程度の広さの人生経験を得てきている大学生が相手の時には無視していい議論だったと思うが、人生経験の乏しい小学生の段階からプログラミングに接する機会を作るとしたら、彼ら彼女らの世界認識への影響も考慮しつつ何を伝えるかを考えていく必要があるだろう、それが本稿で「無限」について言及した理由であった。

8.3　教育者の課題

ここまでの本稿は（乱暴に要約すれば）、古い（逐次型の）枠組みに縛られずに、新しい考え方を取り入れて行こうよという誘いであった。具体的にどんな実習環境が用意できるかといった技術的なものを含めて課題は多々あり、本稿が具体的な提案になっていないことは心苦しい限りである。それに加えて、この最終節ではこれまでの誘いを覆すような指摘を一つして、本稿を終えることとする。

教育者は新しいものとどう付き合うか

教育という営みは変わらず普遍的な内容を扱うべく教程が設計されるのが元来の性質である。勿論、それぞれの学問分野で新たな知見が得られることで変化する内容もあるが[14]、概して不変の知識だと見なして教える側も学ぶ側も扱っていて大きな支障はなかった。ところが、情報教育は、変化の激しい技術の代表格でもあるコンピュータ及びネットワークとの関係が深い。初等中等教育でのカリキュラムは普遍性を担保すべく注意深く構築されているようだが[Mext1]、その内容を実習を通じて習得するためには、現実世界で使われているハードウェアやソフトウェアを使わざるを得ず、その現実世界はドッグイヤーとも形容される速さで変化する世界である。

さらに、例えば歴史や微積分が何の役に立つのか、といった議論を微妙に保留した状態で多くの科目が実利主義から距離を置いた場所で成立しているのに対して、今の情報教育はともすれば「役に立つ」ことが求められがちでもある。その実利圧力もまた、現実世界の変化へのキャッチアップを要求することになる。その結果、情報という科目を扱う教員は、教科の内容についても（本稿によると）知識のアップデートが求められるし、教科の道具についても常にアップデートしていることが求められることになる。これほどに回転速度の速い領域が教科に組み込まれる、これは歴史上初めての事態かも知れない。そうすると、ただでさえ重い負担に苦しんできている教員[Asahi1]が、どうやってこの高速回転する対象世界と（子供達に向き合いつつ）付き合って行くか、これは困難な課題だろう。今、プログラミング教育も情報教育も、難しい曲がり角に来ているようだ。

14) 我々の世代には鎌倉幕府の成立年が「イイクニ」ではなくなった、というのが典型例だが、そういう例が印象に残ることが、逆に見れば内容の変化の度合いが少ないことの証左だと考えていいだろう。

参考資料

[Ted1]: Lera Boroditsky, How language shapes the way we think(Ted Talk),https://www.ted.com/talks/lera_boroditsky_how_language_shapes_the_way_we_think

[Hiraoka1]: Nobuyuki HIRAOKA, 入門用プログラミング言語としてのScala ― 教程と環境 ―, http://id.nii.ac.jp/1025/00001057/

[Masui1]: 増井俊之, 富豪的プログラミング,http://masui.org.s3.amazonaws.com/f/c/fcb2ca45a419206be203c8ca8ed9e02b.pdf

[Web1]: APL 入門, https://fxrobot.hatenablog.com/entry/2013/11/25/162800

[Turing1]: A.M.Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem",http://www.cs.ox.ac.uk/activities/ieg/e-library/sources/tp2-ie.pdf

[Dijkstra1]: Edgar Dijkstra, Go To Statement Considered Harmful,https://homepages.cwi.nl/~storm/teachi

ng/reader/Dijkstra68.pdf
[Mext1]: 文部科学省, 初等中等教育における情報教育,https://www.mext.go.jp/b_menu/shingi/chousa/shotou/056/shiryo/attach/1244848.htm
[Asahi1]: 朝日新聞,「#教師のバトン」 やまぬ大変さ訴える声　文科省は1カ月発信なし, https://www.asahi.com/articles/ASPBQ4QYJPBPUTIL055.html