経営論集　第25巻第1号　2015年　83～104頁

プログラミング教育の問題と対策

小　松　香　爾

1 はじめに

今後、10～20年で人間が行ってきた仕事の47%がなくなるという予測がある。例えばタクシーの運転である。タクシーの運転という仕事は、自動運転技術が成熟し、制度が変れば消滅する。自動運転技術を搭載した自動車の開発は、現在、自動車会社はもちろん、グーグル、アップル、日本ではDeNAといったIT企業でも行われている。

情報分野の変化のスピードは激しい。イノベーションあるいはシンギュラリティと呼ばれるべき変化が数年に1度は起きる。しかし、古い技術が突然のように見直されることもある。最近の例では人工知能である。人工知能のビジネスへの応用は、失敗の連続であった。日本の例としては、1982年から1992年の10年間、経済産業省が500億円以上支出した第5世代コンピュータプロジェクトが挙げられる。第5世代コンピュータプロジェクトの目標は大きく２つあった。前半が、並列推論マシンの実現と、論理プログラミング言語の開発で、後半がその現実社会への応用であった。プロジェクトの結果、ハードウェアと言語は実現できた。しかし、機械翻訳や定理証明、知識推論の実用化はほとんど進まなかった。多くの電機系企業が人的、金銭的資源を出資して参加したが、ビジネスとして利益を出せた企業はなかった。要因はいくつかあげられるが、記憶用デバイスの容量当たりの価格が高く、大量のデジタル化されたデータが存在しなかったのが大きい。また当時は、インターネットも存在せず、データを集めることもできなかった。現在の人工知能で主流となったディープラーニングも、大量の学習用のデータがなければ実現できない。人工知能といっても、コンピュータは思考しているわけではない。確率統計モデルに基づいて、もっとも確からしい解を出力しているにすぎない。

インターネットの高速化と無線化は進んでいる。そして、今後は、センサー技術が成熟し、無線接続のコストが下がり、すべてのモノがインターネットにつながるIOTの時代が来ると予想されている。この流れは、自動車の自動運転が実現するころから加速するはずである。その時、現在とは比較にならないほどの大量のデータ処理が行われるようになるのは間違いない。コンピュータはプログラム次第で、どんなものにもなりうる。しかし、いつの時代でも、データ処理を中心に行われてきた。以前は、人手で処理されていたデータも、ソフトウェアで自動化、あるいは半自動化されている。しかし、要求された処理を行うソフトウェアを自動で書くソフトウェアは開発されることはない。要求は人間の頭の中にあり、それをコンピュータに伝える手段がプログラムを書く以外にはないからである。したがって、プログラムを書くという

仕事はなくなることはない。そもそも、脳はデータ処理装置である。プログラマにならない場合でも、問題解決の方法を抽象化し、厳密にプログラムで記述する訓練は、コンピュータを使わない現実の問題解決でも役に立つはずである。そうであるならば、現在流行している技術の細部を教えるよりも、まずプログラミングを学ばせることが先である。

本論文では、従来のプログラミング教育の問題点を整理、分析する。そして、著者が開発したプログラミング教授法を提案する。提案される教授法ではTAやSAは必須ではない。それにも関わらず、プログラミング教育において頻繁に発生するドロップアウトの問題が生じにくい。

2 「Structure and Interpretation of Computer Programs」の功罪

大学の計算機科学におけるプログラミングの授業で、多くの学生が理解できない事項が2つあるとされている。ポインタと再帰である［Spolsky, 2005］。ポインタと再帰の教科書の決定版として「Structure and Interpretation of Computer Programs」（以下、SICP）［Sussman, 1996］があげられる。SICPは、MITにおける新入生向けの教科書として作られた。SICPではLispの一種であるSchemeがプログラミング言語として採用されている。Lispの構文は非常に単純である。データ構造はリストのみであり、リスト自体の構成要素も、括弧、オペレータ、オペランドしかない。実用性を重視して設計されたCommon Lispはオペレータの種類が多いが、Schemeでは、オペレータの種類は極めて少ない。さらに、SICPでは、consやcar、cdrなどの少数のオペレータしか登場しない。Schemeにあらかじめ組み込まれた関数は極めて少ない。よく使われる機能は、ユーザが関数を作って実現することになる。SICPでも、前半の章で自作する関数が、後半で何度も登場する。

SICPでは、関数の定義および、再帰や並列処理、遅延評価の概念を、Schemeを通じて、初学者にいかに効率よく学ばせるかに焦点が置かれている。したがって、SICPはプログラミング言語の教科書ではなく計算機科学の教科書と見なされ、MITをはじめとする世界の大学の計算機科学部で、初年次における教科書として使用されてきた。しかし、SICPを教科書として使う授業の難易度が高いことも指摘されてきた。ほとんどではないが、かなりの学生がコースをクリアできないのである［Spolsky, 2005］。

SICPの難しさを、計算機科学に向いているかどうかのふるい分けに使えるとして、肯定的に捉えることはできる［Spolsky, 2005］。アメリカの計算機学科では、SICPのコースに挫折した学生が、自らの才能に見切りをつけて他の学科に専攻を変え、幸福な学生生活を送る例がある［Spolsky, 2005］。しかし、日本の大学では転学部や転学科は稀である。制度自体がなかったり、制度があっても事実上は不可能であったり、利用した場合に不利になることが多い。学部学科ごとに入学試験時の最低点が異なるため、学生が入学時に決めた専攻の途中変更はできないか、あるいはできてもレベルが低い学部学科への変更しかできないのである。この点でアメリカの大学とは大きく事情が異なる。また、アメリカにおいても「計算機科学科を卒業する学生が少ない」という産業界からの指摘がある。そもそもLispに代表される関数型プログラミングの知

識は、ビジネスにおいては直接には必要がない。近年、アメリカの大学でも、教育用のプログラミング言語として、SchemeではなくJavaが使用される傾向にある。Javaは命令型言語であるが、CやC++など従来の言語とは異なりポインタが存在しない。Javaでは、参照型の変数が導入され、アドレスの値を代入することができる。しかし、利用者はアドレス値が代入されていることを意識することはない。アドレス値を変更することはできないし、その必要もないからである。ただし、配列などを扱う際にポインタに相当する概念の理解は必要であるが、やはりポインタを直接操作する必要はない。したがって、メモリ領域の確保や開放など、ハードウェアに近いレベルの処理を意識的に行う必要もない。JavaはCやC++と比べて、初学者にとってのハードルが低い言語であると言える。しかし、大学における教育用言語としてのJavaの浸透にも関わらず、アメリカの計算機科学科のドロップアウト率は40%～70%と推測されている［Spolsky, 2005］。

SICPのコースで、挫折する学生が多い理由は、ポインタと再帰の難しさに加えて、関数型プログラミング言語自体のマスターが難しいことが挙げられる。カリフォルニア大学バークレー校では、SICPのプログラムが、命令型言語であるPythonに置き換えられて、授業が実施されている［Hilfinger, 2012］。さらに、このコースでは、内容の大幅な削減も行われている。SICPの4章は、ほとんど含まれず、5章は全く含まれない。つまり、インタプリタの設計に関する内容が大幅に削られ、計算機シミュレータ、およびその上でのアセンブラやコンパイラの実装は全く取り扱われない。

カリフォルニア大学バークレー校は、MITやスタンフォード大学と並ぶ計算機科学の名門であり、世界の公立、国立大学ランキングではトップである。Linuxより堅牢性が高いUNIXであるFreeBSDも、カリフォルニア大学バークレー校の計算機科学学部を拠点として開発された。世界のトップレベルの計算機科学学部でさえ内容が削られ、関数型が命令型の言語に置き換えられてしまうという事実は、SICPの難解さを表している。ただしSICPで取り扱われている事項は、計算機学部のカリキュラムの核となる内容である。最終的に遅延評価やガーベジコレクションができる仮想マシンをつくり、仮想マシン上のアセンブラとコンパイラの実装まで行わせる。それらすべての内容が、Schemeという文法が単純な言語で書かれたプログラムをサンプルに説明されている。したがって、一部の計算機科学の素養があり、かつ熱意もある学生にとっては、最良の教科書であると言える。

SICPが出版されたのは1985年である。日進月歩のコンピュータの世界では1980年代の書籍は古典といえる。それにも関わらず、現在でもSICPを翻訳したり、課題を解いたりするWebサイトやブログが新たに作られている。そして多くの読者が、処理を抽象化する第1章、データを抽象化する第2章で挫折している。仮想マシン上にコンパイラを実装する第4章、第5章にはたどり着けないのである。第4章、第5章に到達できなければ、プログラムがどう解釈され、どう実行されるかというSICPのメインテーマに接することはできない。しかし、2章、3章の内容を理解するためには、理数的な内容、極限や行列、電子回路の知識が必要である。確かに数

値計算のように数学の知識が必要な分野は存在する。対象分野に依存するが、一般的にプログラミングに理数系の知識は必要ではない。また、数学能力とプログラミング能力の相関関係もはっきりしていない。SICPを読むために、数学を勉強しなければならないというのは、手段の目的化である。教育において手段が目的化してしまうと、到達目標にブレが生じ、学習者の意欲を削ぐことになる。SICPの教科書としての有効性は、使用される場の環境に大きく依存するといえる。

3　FizzBuzz問題の有用性

SICPを最後まで理解できる学生にプログラミングの素養があることは疑いない。しかし、SICPに含まれる課題よりはるかな単純なプログラムを、計算機科学部の卒業生の過半数が書けないことが指摘されている［Ghory, 2007］。その問題はFizzBuzz問題と呼ばれているものである。

FizzBuzzは、1から100まで数え始めて、3で割り切れるときは”Fizz”を、5で割り切れるときは”Buzz”を、3で割り切れてかつ5で割り切れるときは”FizzBuzz”を、どれにも当てはまらないときは数字をそのまま出力するという問題である。FizzBuzzは、現実世界でも行われてきた言葉遊びであり、問題と呼ぶには単純すぎる。しかし、IT企業において、プログラマの採用時に適性があるかどうかを判定するテストとしても使われた。通常、FizzBuzzを解くためのプログラミング言語は、何を使っても良い。例えばJavaを用いて、トリッキーなコードを使わずに書いた場合は、以下のようなコードになる。

```
class FizzBuzz {
  public static void main(String[ ] args){
    for(int i=1; i<=100; i++){
      if (i%3==0 && i%5==0) System.out.print("FizzBuzz ");
      else if (i%3==0) System.out.print("Fizz ");
      else if (i%5==0) System.out.print("Buzz ");
      else System.out.print(i+" ");
    }
  }
}
```

上記のように、FizzBuzzは10行で書け、難しいロジックは含まれない。しかし、アルゴリズムが全くないわけではない。繰り返し処理の中で、条件分岐させる必要があるからである。構造化プログラミングの基本3構造は、順次、選択、繰り返しである。命令型言語でFizzBuzzを書けば、3構造を組み合わせて使うことになる。したがって、プログラミングの初心者にとっ

ては、ハードルが高い可能性がある。

FizzBuzzのような基本的な処理を書けたとしてもプログラマとは言えない。しかし、FizzBuzzが書けなければプログラマとは言えない。つまりFizzBuzzが書けることはプログラマであるための十分条件ではないが、必要条件であると言える。それにもかかわらず、現実には、FizzBuzzが解けない現役の職業プログラマが存在する。現実世界では、簡単なゲームとして実施されるFizzBuzzだが、プログラムとして厳密に記述するのは意外と難しいということになる。それゆえ、FizzBuzz問題は、世界中の大学の計算機科学学部やIT企業で、広く知られることになったのである。

日本のソフトウェア業界では、基本的に人月計算で見積もり書が作成される。そして、開発プロジェクトの現場では、納期直前にプログラマの徹夜が連続する事態が頻繁に生じてきた。いわゆるデスマーチである。デスマーチの発生は、ユーザ企業側の開発途中での仕様変更が、主な原因されている。しかし、FizzBuzzも書けないような質の低すぎるプログラマが、現場に投入されることも大きな原因であると推測される。プロジェクトでデスマーチが発生、あるいは発生しそうになると、追加のプログラマが投入されることもあるが、投入されるプログラマも質が低い場合が多い。炎上プロジェクトに投入されるようなプログラマは、下請け会社、あるいは実質的には人材派遣会社と変わらないIT企業に所属しているケースが多いからである。したがって、ソフトウェア開発プロジェクトで繰り返される悲劇を回避するためには、プログラマ全体の底上げより、最低限の質保障がされたプログラマを多く養成することの方が現実的である。プログラマの質保障の基準としてFizzBuzz問題は適しているといえる。基本3構造を理解していなければ解けないが、現実世界では子供でもできる単純な問題だからである。

文京学院大学経営学部における「プログラミング言語」の授業では、半期でJavaの演習が行われる。Javaの演習がひと通り終わった後に、FizzBuzz問題とよく似たチェック用の問題を出題する。チェックの目的は学生の到達度を測定するためである。しかし、チェックの結果は、例年、芳しくない。年度にもよって違いはあるが、正解を書ける学生が1割いれば良いという程度である。さらに、この数字は「教科書を参考にしてもよい」とした上のものである。教科書を見ないで書ける学生はほとんどいない。理系の大学に所属している留学生や、日本の理系の大学から3年次に編入してきた学生ぐらいである。なお、「プログラミング言語」は3年次に開講される授業である。経営学部では、他にプログラミングの授業として２つの科目が用意されている。１つはVisual Basicを用いたアプリ作成の科目であり1年次に開講される。もう１つはJavaScriptを取り入れたWebページを制作する科目であり2年次に開講される。これら２つのプログラミング関係の科目を履修済みでかつ「プログラミング言語」を受けた後に、1割程度しかFizzBuzzを解けないというのであれば、プログラミング教育の効果が全くあがっていないことになる。大学教育の質保障の観点からしても、大きな問題である。ただし、これら３つのプログラミングが含まれる科目は、いずれも選択科目である。したがって、「プログラミング言語」を履修する学生の中には、プログラミング自体が全く初めてという学生も存在すると推測され

る。現状では、学生がプログラミング関係の授業をどれだけ受けた上で、FizzBuzz問題に対面しているかが把握できない。大学全入時代において、15コマのプログラミングの授業を受けただけでFizzBuzzのプログラムを書けるようになるというのは、教員側の甘い想定だと言わざるをえない。

Javaは、言語仕様が巨大であり機能も豊富で覚えるべき事が多く、そもそも教育用の言語として向いているのかという問題もある。しかし、それはVisual BasicやJavaScriptでも事情は変らない。「プログラミング言語」では、柴田望洋著の「明解Java入門編」がテキストとして用いられる。履修条件を設定しない半期15コマの授業であるため、教育内容は絞らざるをえない。授業の範囲は、文字列の表示、変数、代入、条件分岐、繰り返し、配列、メソッド、クラスである。授業形式は、プログラムを教員がリアルタイムで入力しながら、Javaの文法や構文の意味を解説するという特殊な演習形式である。教員は、教科書のソースコードをほとんど見ずに入力する。学生の理解を深める、あるいは復習や予習のために、教科書に書かれていないコードを入力することも頻繁にある。したがって、勘違いやミスも起き、コンパイルエラーと実行時エラーの見方や、バグのつぶし方も実演されることになる。このような授業形式を取ることの意図は、プログラミングへのハードルを低くし、恐怖心を薄れさせることにある。医療や介護などとは異なり、プログラミングにおいて間違いは致命的ではなく、間違えていても良いのでとりあえずやってみるという姿勢が重要であるということを伝えたいのである。この形式の授業の特徴は、意欲に乏しい学生でも、プログラミングの経験を得られることである。エディタでプログラムを書き、コンパイルし、間違いを訂正し、プログラムを動かすという一連の手順は身につけることができる。手順を身につけるだけでは、プログラムを書けるようにならないことは自明である。事実、FizzBuzzのプログラムでさえ、ほとんどの学生が書けるようにならない。しかし、プログラミングする際の姿勢とコンパイルから実行までの手順を身につけさせることは、将来、自習しようとするときの橋頭堡となりうる。

4　プログラミングの素養

プログラミングは文章を書くのに似ている。単語を文法通りに並べただけでは文章にはならないのと同様に、表記法や文法を知識として覚えてもプログラムは書けない。いずれも、抽象的な思考能力と、書くための訓練が必要である。整然とした文章が書ける学生の方が、プログラムを書けるようになりやすいことは間違いない。

しかし、文章を全く書けるようにならない人はほとんどいないのに対して、プログラムを全く書けるようにならない人は沢山いる。この違いは、自然言語と人工言語の違いであり、会話など日常生活における目に見えない訓練、要求される厳密性など様々な要因から生じている。しかし、もっと大きな違いは、プログラムは表現の手段ではあるが、問題解決の手段でもあるということである。機能が実現され、動作して、意図通りの結果が出なければプログラムとは呼べない。趣味のプログラミングを別にすれば、プログラムを書くこと自体は決して目的には

なり得ない。実現されるべきなんらかの目的の為にプログラムは書かれるのである。

プログラムが書けるようになるか否かは、モデルを心の中に作ることができるかによるという学説がある。「The camel has two humps」で、プログラミングの習得における、学生のメンタルモデルの重要性が提示されたのである［Bornat, 2006 ］。この論文のタイトルの意味は、「プログラミングができるグループとできないグループは、２つの独立な正規分布に分かれる」である。

プログラミングの習得においては中間層が存在せず、どの大学の計算機科学学部でも、30%～60%の学生が最初のプログラミング科目の単位を落とすとされている。また、プログラミング教育には3つのハードルが存在するとされている。1つは代入と逐次実行、２つ目は再帰と繰り返し、3つ目は並列処理である。Dehnadi と Bornat の実験では、代入と逐次実行のみに焦点が当てられている。これらは、再帰、繰り返し、並列処理よりも容易であるという理由からである。したがって、この実験でテストされるメンタルモデルは、チューリングマシンの計算モデルのごく一部にすぎない。

図 1が実際に学生に課されたテスト問題の例である。初回のテストはコースの最初、2回目のテストはコースの第3週に実施された。3回目のテストも予定されていたが、実際には実施されなかった。2回目までのテストで、学生のプログラミング適性を測定できたと判断されたからである［Bornat, 2006］。

1. Read the following statements and tick the correct answer in the front column.

```
int  a = 10;
int  b = 20;

a = b;
```

The new values of a and b are:

- ☐ a = 30 b = 0
- ☐ a = 30 b = 20
- ☐ a = 20 b = 0
- ☐ a = 20 b = 20
- ☐ a = 10 b = 10
- ☐ a = 10 b = 20
- ☐ a = 20 b = 10
- ☐ a = 0 b = 10
- ☐ **If none, give the correct values:**
  a = b =

図 1　A sample test question

出典「The camel has two humps」p.6

図 1のテスト問題の正解は、通常の命令型プログラミング言語では、a=20, b=20である。しかし、このテスト問題では、学生の解答が正解かどうかは問題ではない。代入の逐次実行のテスト群が提示されたときに、学生が一貫したメンタルモデルを保てるか否かがテストされるのである。図 2には、予想される学生のメンタルモデルが示される。1回目のテストの結果、44%の学生がすべての、あるいはほとんどすべての問題に対して一貫したメンタルモデルを保てた。39%の学生は一貫したメンタルモデルを保てなかった。8%の学生は無回答であった［Bornat, 2006 ］。

1. Value moves from right to left (a := b; b := 0 – third line in figure 1).
2. Value copied from right to left (a := b – fourth line of figure 1, and the 'correct' answer).
3. Value moves from left to right (b := a; a := 0 – eighth line of figure 1).
4. Value copied from left to right (b := a – fifth line of figure 1, and a reversed version of the 'correct' answer).
5. Right-hand value added to left (a := a+b – second line of figure 1).
6. Right-hand value extracted and added to left (a := a+b; b := 0 – first line of figure 1).
7. Left-hand value added to right (b := a+b – omitted in error).
8. Left-hand value extracted and added to right (b := a+b; a:=0 – omitted in error).
9. Nothing happens (sixth line of figure 1).
10. A test of equality: nothing happens (fourth and fifth lines of figure 1).
11. Variables swap values (seventh line in figure 1).

図 2　Anticipated mental models of assignment

出典「The camel has two humps」p.7

Dehnadi と Bornat の実験では、学生のメンタルモデルとプログラミングの成績の関係が調べられた。授業内試験は、プログラミングの到達度を測るテストであり、第7週と第11週に行われる。テストの内容は図3～図5のようなプログラミング言語に関する基本問題である。

Consider the following program fragment.

```
if (mark > 80) grade = 'A';
else if (mark > 60) grade = 'B';
else if (mark > 40) grade = 'C';
else grade = 'F';
```

What is the effect of executing this code if the variable `mark` has the value -12?

(a) The program will crash.

(b) The program will output an error message.

(c) The variable `grade` will be undefined.

(d) The program will never terminate.

(e) The variable `grade` will be assigned the value 'F'.

図 3　A technical multiple-choice question from the week 7 in-course exam

出典「The camel has two humps」p.8

図3の問題は、単純な if 文による分岐である。C か Java のプログラムの一部を抜き出したもの

であり、極めて簡単な問題である。markの値が-12のときmark>80, mark>60, mark>40はいずれもfalseになる。したがって、gradeには文字Fが代入され、正解は(e)である。

What is wrong with the following main method which is intended to add the numbers from 1 to 4 together and print out the result which should be 10? Give a correct version:

```
public static void main (String[] args) {
   int total = 0;
   for (int i = 1; i < 4; i++)
      total = total+i;
   System.out.println("Total:  " + total);
}
```

図 4　A technical write-in question from the week 11 in-course exam

出典「The camel has two humps」p.9

図4の問題は、単純なfor文による繰り返しである。図3までの問題とは異なり、Javaのメインメソッド全体が出題されているが、やはり簡単な問題である。1から4までの整数を足した値が10になるようにするためには、プログラムの3行目の「for（int i=1; i<4; i++)」を「for（int i=1; i<5; i++)」と訂正する必要があり、それが答えである。

The following code is an incomplete version of a 'guess the number' program, in which the user is repeatedly invited to guess a hidden number. Add the necessary code to the body of the while loop to check the user's guess and output appropriate messages depending on whether or not the guess is correct.

```
final int HIDDENNUM = 20;

String numStr =
   JOptionPane.showInputDialog("Enter your guess (zero to finish)");
int guess = Integer.parseInt(numStr);

while (guess != 0) {

   //ADD YOUR CODE HERE

   String numStr =
     JOptionPane.showInputDialog("Enter your guess (zero to finish)");
   int guess = Integer.parseInt(numStr);
}
```

図 5 A technical creative question from the week 11 in-course exam

出典「The camel has two humps」p.10

図5の問題は、while 文の中で、if 文を使って条件分岐させる問題である。FizzBuzz の類似問題といえるが、条件分岐はFizzBuzz より条件式が簡単で分岐も少ない。Else 文を使っても、正解がかけるが、使わずとも以下のように書ける。

```
if (numStr == HIDDENNUM){
  System.out.println ("correct");
  break;
}
System.out.println ("incorrect");
```

メンタルモデルの結果と二回の授業内試験の点数の平均点との関係は、図 6に示される。1回目のテストで一貫したメンタルモデルを形成できた学生の人数は黒の棒グラフで、メンタルモデルを形成できなかった学生の人数は白の棒グラフで表されている［Bornat, 2006 ］。

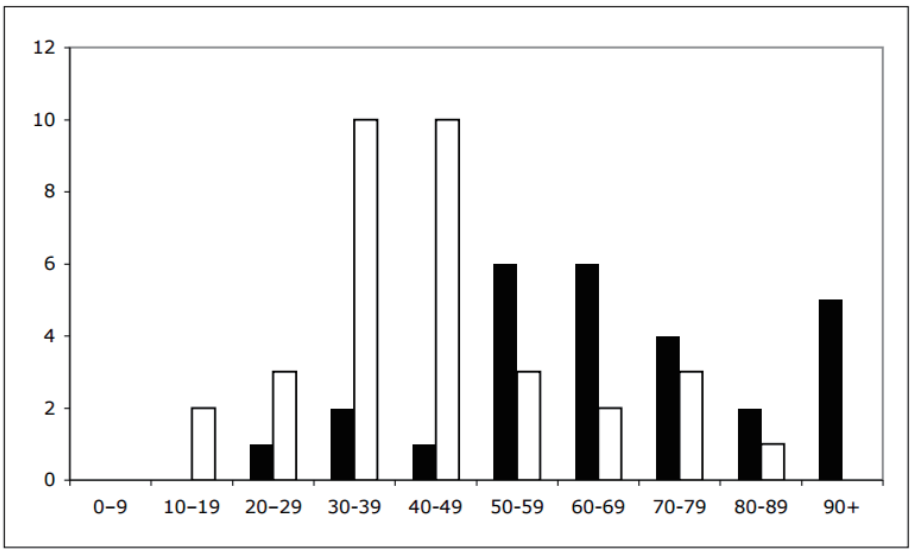

図 6　First-test consistent（black）and inconsistent/blank（white）average in-course exam results

出典「The camel has two humps」p.13

図7 では、1回目のテストでメンタルモデルを形成できなかったが、2回目のテストで形成出来るようになった学生の人数が、灰色の棒グラフで表されている。なお、一回目のテストでメンタルモデルを形成できた学生の人数が黒で、二回目のテストでもメンタルモデルを形成出来なかった学生の人数が白で表されている［Bornat, 2006 ］。

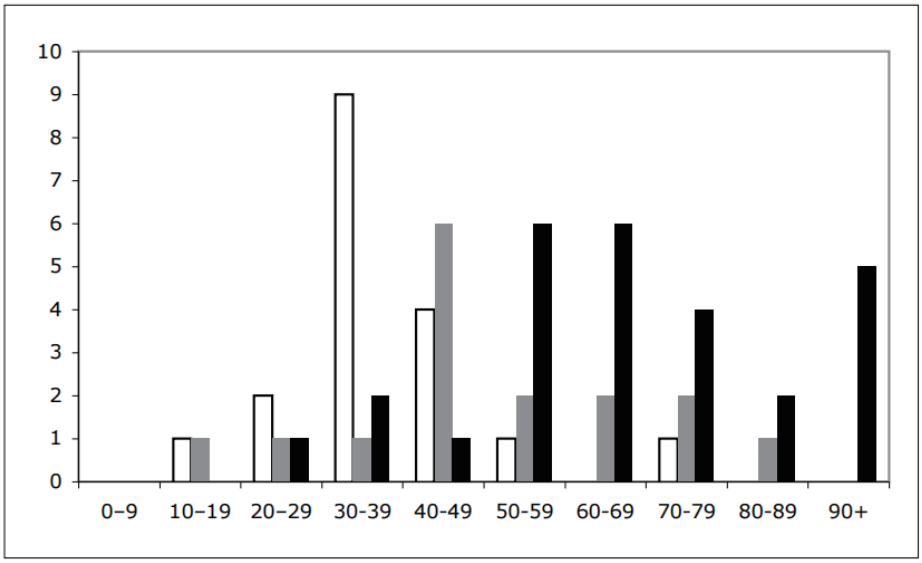

図 7　Second-test inconsistent（white）, second-test consistent switchers（grey）and first-test consistent（black）average in-course exam results

出典「The camel has two humps」p.13

Dehnadi と Bornat の実験の対象となったコースでは、最終的に61人中32名が単位を落とした。しかし、1回目のテストで一貫したメンタルモデルを形成できた学生は、27人中6人が単位を落としたにすぎなかった。1回目のテストは、プログラミングの授業を行う前に実施される。図7の灰色の棒グラフが、授業を受けた後に一貫したメンタルモデルを形成できた学生の人数を表している。プログラミングの素養が、授業で養われたグループといえる。しかし、このグループの授業内試験の成績は、最初から一貫したメンタルモデルを形成できた学生と比較すると明らかに悪い。プログラミングの素養は、1回目のメンタルモデルのテストで、ほぼ判定できることがわかる［Bornat, 2006］。

5　プログラミング教育の理想と現実

プログラミング教育の工夫を実践し、その有用性を示した論文は多い。しかしDehnadi と Bornat の実験のようにプログラミングの素養を調べた論文は存在しない。「生まれつき、あるいは大学に入学するまでに培われたセンスが重要であり、教育による伸びは期待できない」ということが、Dehnadi と Bornat の論文では示唆されている。プログラミングの素養が伸ばしにくいという問題への対策は2つ考えられる。1つは、義務教育の段階でプログラミング教育を行うことであり、もう1つはプログラミングの授業を取る前にDehnadi と Bornat のようなテストでふるい分けを行うことである。

後者のふるい分けは、義務教育では実行できない。義務教育の性格上、履修制限を設けるこ

とができないからである。しかし、大学教育では実行できる。現状、ほとんどの大学において、コンピュータ関係の科目は、コンピュータの台数で履修制限がかけられている。コンピュータの台数を履修希望者がオーバーした場合は抽選になる。文京学院大学の経営学部のコンピュータ科目では、学年が上の学生を優先的に受講させることが慣習となっている。このしくみは、学年が上の学生ほど将来における履修の機会が少ないことを考慮しているという点で合理性がある。しかし、教育の質保証の観点からすると、好ましくない。プログラミングの素質のない学生が受講することになり、教員は説明のレベルや授業の速度を上げられなくなるからである。プログラミングの素養がないだけならばまだ良い。卒業のための単位が欲しいだけの学生が履修登録をして、授業についていけず、途中でドロップアウトするケースが頻繁に見られる。途中から貴重なコンピュータ演習の席は空くことになる。本来ならば、下の学年でプログラミングを学びたかった素質も意欲もある学生が座るべき席である。したがって、本来、プログラミングの授業がある学部学科では、DehnadiとBornatが実施したような簡単なテストを、初年次の履修登録の前に行うべきである。

前者は、小学校での英語教育の導入と同様の根拠に基づく。接する時期を早く、また接する時間を長くすれば、素養がなくても身につくというものである。プログラミング言語は人工言語とはいえ、言語には違いはない。したがって、英語教育同様に幼少期からの慣れがその後の伸びにプラスの影響を及ぼす可能性は大きい。実際、義務教育でのプログラミング教育は、2012年から施行された中学校学習指導要領で部分的に実現されている。この新学習指導要領では、技術家庭科の中に「プログラムによる計測・制御」が必修として盛り込まれている。具体的な内容は、「コンピュータを利用した計測・制御の基本的な仕組みを知ること」と「情報処理の手順を考え，簡単なプログラムが作成できること」となっている。しかし、このような内容を、技術家庭科の一部として教えることは時間的に無理がある。大学における90分15コマのプログラミングの授業を受けた後でも、1割程度しかFizzBuzz問題のプログラムを書けるようにはならないのである。プログラミングまで教えるためには、コンピュータ科目として技術家庭科から独立させる必要がある。

しかし、コンピュータ科目として独立した科目にするためには、大きな障害がある。教員免許制度の問題である。高校の教員免許に「情報」が存在するのに対して、中学には「情報」が存在しない。プログラミングを教えることができる教員が確保されていない現状では、義務教育におけるプログラミング教育は絵空事でしかない。プログラミング教育における人材不足の解決には、産官の働きが不可欠である。政府にもその認識はある。「産学官連携による実践的 IT 人材を継続的に育成するための仕組みを構築し、義務教育段階からのプログラミング教育等のIT教育を推進する」という文言が、安倍政権の成長戦略に盛り込まれている［日本経済再生本部, 2015］。しかし、2001年のIT基本法でも、「国民の情報活用能力の向上及び専門的人材の育成」が施策の基本方針として挙げられていた。それにも関わらず、いまだに中学の教員免許に「情報」がないのである。

時代の変化に公的な教育が追従できないのは、海外でも事情は同じである。結果として、従来の教育機関に頼らない、子供向けのプログラミング教育の動きが生じた。その1つが2011年にアイルランドで始まったCoderDojoである。CoderDojoの対象は7～17歳で、目的はプログラミングの基礎を学ばせることである。CoderDojoの運営や指導者はボランティアであり、参加者から参加費を取ることを禁止している。ハードウェアはBYOD（Bring Your Own Device）であり、指導者と場所の確保さえできれば開催できる。開発ソフトは、MITのミッチェル・レズニックによって、子供向けの教育用言語として開発されたScratchが使われる場合が多い。CoderDojoは欧州を中心として世界各地の多数の地域で開催されており、2012年6月には東京で、アジア初の「CoderDojo Tokyo」が開催された。2015年7月までに、59カ国の750カ所に道場がある。

プログラミング教育の先進国であるアメリカにおける子供向けプログラミング教育の動きもいくつかある。NPOのCode.orgがその代表例である。Code.orgは2013年の1月にプログラミングを全米の教育カリキュラムに組み込むこと設立の目的として設立され、2月にプログラミング教育のためのサイトを公開した。Code.orgのサイトでは、オバマ大統領、マイクロソフト創業者のビル・ゲイツ、アップル創業者のスティーブ・ジョブズ、フェイスブック創業者のマーク・ザッカーバーグ、ツイッター創業者のジャック・ドーシー、ドロップボックス創業者のアンドリー・ヒューストンらが動画に登場し、プログラミングの重要性を訴えている。Code.orgの使用言語はScratchによく似たblockyと呼ばれるブロック配置型のビジュアルプログラミング言語である。Code.orgが実施した1時間のプログラミングチュートリアルサイト「Hour of Code」は、5日間で1500万人がユーザ登録した。また、ユーザによって書かれたコードは6億行に達した［Code］。CorderDojo、Code.orgいずれも、プログラミング言語として、教育用言語、それもビジュアルプログラミング言語を採用しているのが特徴である。子供を対象にした教育では、エディタでコードを入力させるのはハードルが高いという見解からであると推測できる。Scratchを開発したミッチェル・レズニックの主張は、「想像し、作り、遊び、共有し、振り返り、想像に戻る」という幼稚園における学びのスパイラルが、21世紀の教育に適しているということである［Resnick, 2007］。現代におけるい教育は、全ての年齢において、創造的思考力をのばすことを助けるものであり、Scratchというビジュアルプログラミング言語は、図8のスパイラルを回す際に使うツールとして開発された。Scratch1.4ではプログラムの制作環境は、ダウンロードしてローカルで実行されるだけであった。しかし、Scratch2.0では、インターネットのScratchサイトで作成されたプログラムが共有までされる。共有されたプログラムにはYoutubeと同様なコメントがつくため、図8における「share→refrect→imagine」が促されることになる。

図 8　The Kindergarten approach to learning
出典「All I Really Need to Know（About Creative Thinking）I Learned
（By Studying How Children Learn）in Kindergarten」p.2

通常のプログラミング言語では、エディタでソースコードを入力して、コンパイル、エラー修正、実行、デバッグという手順が必要である。図8の学習サイクルを回す以前に、言語の習得自体が目的になってしまいかねない。

最初に学習させるプログラミング言語は、何が適切なのかという議論は昔からある。日本では、1980年代ではBASICが定番であった。多くの8bitパソコンで、電源を入れた後、ROMからBASICのインタプリタが起動するという環境面の利便性が大きい。このROM-BASICには、簡易OSとしての役割もあった。BASICの命令で、外部記録装置からプログラムをメモリにロードしたり、メモリのプログラムを外部記録装置にセーブしたりしていたのである。当時のパーソナルコンピュータはホビーユースが中心であった。日本では1982年にBASICマガジンが発行された。BASICマガジンには、ゲームのソースコードが多数掲載されていた。そして、主に小中学生によってコードが入力されていた。当時は任天堂のファミリーコンピュータが存在しなかったからである。現在は、BASICが使われる場面は少なくなった。しかし、プログラミング初学者用にマイクロソフト社が「Small Basic」を公開している。Small Basicの統合開発環境は通常のGUIアプリケーションであるが、Eclipseなどと比べると、極めてシンプルである。「センター試験用手順記述標準言語」としてもBASICが採用されている。2015年度のセンター試験、旧数学II・旧数学Bの第6問である。

[プログラム1]

```
100 INPUT M
110 INPUT N
120 LET Q=INT (M/N)
130 PRINT Q
```

```
140 LET R=M-Q*N
150 LET M=N
160 LET N=R
170 IF R>0 THEN GOTO 120
180 END
```

[プログラム2]

```
100 LET M=1
110 LET N=1
120 INPUT Q
130 IF Q=0 THEN GOTO 180
140 LET R=N
150 LET N=M
160 LET M=Q*N+R
170 GOTO 120
180 PRINT M:"/":N
190 END
```

プログラム1、2共に、［　　］の中に入る式を選択肢からマークさせる問題である。BASICでは、先頭の行番号が付けられる。そして、行番号は、そのままラベルとして使うことができる。大規模なプログラムの場合、goto文の多用は混乱の元になった。しかし、小規模なプログラムならかえって理解しやすいと言える。基本的に命令は上から下に逐次実行され、例外的にgoto文で前に戻ったり、先に飛んだりするということが表現されているからである。比較のため、プログラム1と2をJavaで書くとそれぞれ以下のようになる。

[プログラム1]

```
import java.util.Scanner;
class Program1 {
  public static void main (String[] args){
    Scanner stdin = new Scanner (System.in);
    int m = stdin.nextInt ();
    int n = stdin.nextInt ();
    int q, r = 0;
    while (r>0){
     q = m/n;
```

```
        System.out.println (q);
        r = m ? q*n;
        m = n;
        n = r;
    }
}
```

[プログラム2]

```
import java.util.Scanner;
class Program2 {
  public static void main (String[] args){
    Scanner stdin = new Scanner (System.in);
    int m = 1;
    int n = 1;
    int q, r;
    do {
    q = stdin.nextInt ();
      if (q==0) break;
  r = n;
    n = m;
    m = q*n + r;
    } while (true);
    System.out.println (m+"/"+n);
  }
}
```

上記のJavaのプログラムはブロックを表す{}とインデントで構造化されている。しかし、この程度の長さのプログラムでは、構造化することのメリットはなく、初学者にとってはハードルになる。また、初学者向けのプログラミング教育で、長いアルゴリズムを実装させることもない。また、構造化されていないBASICの方が、アセンブリ言語に近く、コンピュータの動作を直接的に表している。アセンブリ言語の入門としてもBASICの方が適していると言える。実際、1980年代には、BASICの熟練者の中には、プログラムの実行スピードを追求して、アセンブリ言語に移行する者もいた。

いかなるCPUも、機械語でしか動作しない。したがって、いかなるプログラミング言語で組まれたプログラムも、最終的には機械語で実行される。Cコンパイラはソースコードを機械

語に変換する。ただし、Javaでは、ソースコードが機械語に変換されるわけではない。Javaのソースファイルは、Javaコンパイラによってバイトコードと呼ばれる中間コードに変換される。バイトコードは、Java仮想マシンで解釈される。OSごとに仮想マシンがインストールされていれば、共通のバイトコードでコンピュータが動くことになり、「write once, run anywhere」が実現されることになる。それに対して、アセンブリ言語は機械語の命令とアセンブリ言語のニーモニックは1対1対応である。アセンブリ言語でプログラムを組めるということは、コンピュータを直接、機械語でコントロールできるということに等しい。

現在、プログラムがアセンブリ言語で書かれることはほとんどなくなった。日本の大学、特に電子、情報系の学科ではアセンブリ言語が教えられることがある。プログラミング教育というよりも、コンピュータの構成やCPUによるハードウェアの制御を理解させる目的であることが多い。CPUの設計にも依存するが、一般的にアセンブリ言語の命令は単純で覚えやすい。しかし、実用的なプログラムを書くには、人間の負担が大きすぎる。

マイクロソフト社は、学習コストが低い教育用プログラミング言語として、Small Basicを公開している。Small Basicは命令の数や引数が、Visual Basicに比べて大幅に削減されている。統合開発環境も1980年代のROM-BASICを彷彿とさせるほどシンプルである。

6　変数の壁

変数はプログラムの中でなんらかの値を保持するためのもの、正確にはメモリのある領域に名前をつけたものである。変数は、アセンブリ言語では、レジスタおよびメモリ領域とメモリアドレスの組み合わせに相当する。したがって、実用的なプログラムを作るまでのプログラミング言語の解説書、特に入門書では、しばしば「箱」に例えられる。ただし、「メモ用紙」の方がアナロジーとして適切であるという見解もある［前橋和弥, 2004］。その理由は、変数への値の代入が、箱より正確に表現できるというものである。

箱を用いた説明では、箱の中身を、他の箱に入れた場合に、元の箱から入れた箱に中身が移動する。その結果としてもともと中身が入っていた箱が空になってしまう。空となることが、実際の変数の代入の際に起きる挙動とは異なる。箱ではなくメモならば、新しいメモに記録を移しても元のメモの記録は消えないので、代入の挙動をより正確に表せるということである。

しかし、メモは変数を教える際の例えとして優れているとは言えない。記憶領域であるメモリとの違いがほとんどないからである。結局のところ、変数の正確な理解にはハードウェアの知識が必要である。しかし、変数のハードウェアレベルの理解は、現在のプログラミング教育では必要ないといえる。重要なのは、変数宣言や変数への値の代入の際に、メインメモリの状態が変化することである。これは「プログラムがなんらかの状態をコンピュータに記憶させる」と教えればよい。

データ型、スコープ、ローカル変数とグローバル変数の違い、仮引数と実引数の違いなど、変数がらみで理解しにくい事項が多い。しかし、変数そのものと変数への値の代入を理解でき

ない学生は少ない。ただし、変数を理解することと、変数を使用してプログラムを組むことの間には厚い壁が存在する。その壁を突破させるには、ゲームを例にして、変数の必要性を説明するのが良い。ほとんどの学生が知っている例として、スーパーマリオブラザーズを挙げる。スーパーキノコを取ったときにマリオは無敵になる。スーパーキノコを取った直後に、マリオの状態を保持する変数にそれまで入っていたのとは違う値、すなわち無敵に相当する状態を表す値が代入されるからである。無敵状態では、敵との当たり判定で通常とは違う処理が行われる。通常ならば、敵と当たるとマリオが死ぬが、無敵状態ならば敵が死ぬ。無敵状態のような分岐は、if文などで状態を表す変数の値を調べれば実現される。変数の使用例としては、ドラゴンクエストのようなロールプレイングゲームでも良い。船を手に入れた後には海に入れるが、船を手に入れる前では海には入れない。このとき、船を持っているかどうかという状態を記憶する変数が必要である。

状態は変数で保持され、状態が変れば変数の値が更新される。状態の変化はユーザの入力、時間の経過、繰り返しの回数、モードの変更など様々な要因で起きる。理解しやすい実例を用いて、変数を使うメリットや変数は何を実現するものなのかを説明する必要がある。なぜなら、命令型プログラミング言語において、変数を使えない学生は、実用的なプログラムは書けないからである。条件分岐や繰り返しでも変数が必須である。

scratchなどのビジュアルプログラミング言語では、命令や文を表すブロックが使用される。ブロックを組み立てることでプログラムを組む。マウスでの操作が基本簡単なプログラムなら変数を意識的に使う必要がない場合もある。しかし、なんらかの状態を保持、変更する場合には、やはり変数を宣言する必要がある。通常の命令型プログラミング言語と同様に、文字列を入力して、変数宣言するのである。命令型言語においては、変数をどう使うかが第一関門である。変数を教えていない状態では、条件式や制御変数を教えられないため分岐や繰り返しを教えられない。しかし、分岐や繰り返しを教える際に、変数の必要性の理解を深められるともいえる。

7　ポジティブな自己フィードバックを重視したプログラミング教授法

CoderDojoやcode.orgと教育機関でのプログラミング教育の違いは、後者にはシラバスと学習者を評価する必要があることである。CoderDojoでは、シラバスがないので到達目標もなく、それゆえドロップアウトもない。大学ではセメスターが始まる前にシラバスを書き、セメスターが終われば、学生を評価した上で単位認定を行う。必修科目であれば、難易度の設定に注意を払う必要がある。難易度を高くしすぎると学生の出来が悪く、留年者を大量に出しかねない。しかし、そのような場合でも、教員は相対評価を導入して、少数の学生しか留年させないという手段も選択できる。問題なのは、むしろ選択科目の場合である。難易度を高くしすぎたり、内容がつまらなすぎたりした場合は、セメスターの途中で脱落する学生が続出する。学生に「プログラミングは難しい」という意識だけを持たせて、以降の人生におけるプログラミングの学

習意欲を削いでしまう。これはプログラミングだけでなく、英語や数学などの科目にも同じ事が言える。しかし、英語や数学を嫌いになる、あるいは苦手意識を持つのは、義務教育で学習してからである。プログラミングの場合、現状、義務教育でほとんど学習の機会がない。大学入学以降も、文系の学部はもちろん、理系学部でさえ、カリキュラム上で大きく時間を割かれていることはない。従来型のプログラミング教育で単位を取るためには、心の中に一貫したモデルを保持することができるという素養が学生に必要である［Bornat, 2006 ］。しかし、そのようなモデルを作ることができない学生に、いかに教えれば良いかということは示されていない。素養は素養であり、教育で養成することはできないという見解に、同意するところはある。しかし、素質がない学生でも、それなりに簡単なプログラムを組めるところまでは持っていき、そこで単位を付与するというのが、経営学部でプログラムを教える教員の義務であるという思いは常にある。

本論文では、学生に苦手意識を持たせないプログラミング教授法を提案する。山形大学工学部で7年間、文京学院大学経営学部で11年間、試行錯誤して到達した教授法である。本教授法の特徴は主に以下の6つである。現在も、文京学院大学経営学部におけるプログラミングのゼミで、TA や SA なしで実施されている。

1. 題材は90分以内に完成させることができるゲームあるいは、ゲームの断片とする
2. 教員がリアルタイムで、教科書を見ずにプログラムを入力し、センターモニタに表示する
3. クリッカーを使用して、学生の入力状況を常にチェックする
4. プログラムの説明は、入力時に同時に行う
5. プログラムを完成させた後に、クリッカーで学生の理解度を確認する
6. 理解度が低い場合には、ほぼ全ての行に対しコメント文を入れる
7. 学生の最終的な理解度に応じ、次のプログラミングの題材を決める

本教授法のプログラミング言語は、Flash の ActionScript2.0を採用している。ActionScript は、もともとはアニメーションを制御するために作られたスクリプト言語である。Flash は動画作成ソフトであるが、Flash がそのまま統合開発環境でもある。したがって、ゲームやインタラクティブムービーを作るのには向いている。ActionScript の特徴は、ゲームなどのプログラムを作る際に、グラフィックスライブラリを使用しないというところにある。グラフィックスやムービーは、Flash で作成すれば良いからである。その点では、プログラミングにブロックを使う Scratch に似ていると言える。Scratch と違う点は、エディタにソースコードを入力する必要があるところである。

上記の1.において題材をゲームにする理由は、ほぼ全ての学生が学習目標を明確に理解でき、また、興味を少しは持てるからである。 課題のための課題を積み重ねて、最終的に電卓アプリを作れるようになると言われても学生のモチベーションが上がることは稀である。興味があ

り実現したいものを学習の対象にしたほうが、学習効果が上がる。その上で、学習内容が将来的に何に使えるか、まずビジョンを示す必要がある。適切なビジョンを示すことが教師の最初の役割である。日本の教育は、何に使えるかをまったく教えないで、ひたすら基礎を勉強させる傾向がある。学生のモチベーションを高めることが軽視されている。現在、その反省として、アクティブラーニングの導入が、文部科学省を中心に進められている。アクティブラーニングの様々な手法自体は、試行錯誤の結果洗練されてきたもので、もちろん良いものである。しかし、適切な動機付けなしにアクティブラーニングの手法を導入しても効果は期待できない。アクティブラーニングとはいえ、学習するのは学生自身だからである。なんらかの目標へ到達するためには、もちろん基礎固めが必要である。しかし、基礎固めそれ自体を目標にしてはならない。ほとんどの学生にとって基礎固めはつまらないものである。ほぼ全ての学生にとって興味のある題材はやはりゲームになる。

上記の2.において、教員がリアルタイムで入力するのは、学生に緊張感をもたせつつ、プログラミングのハードルの低さを示したいからである。テキストがあるとどうしても後で見て入力すれば良いという雰囲気が生じてしまう。また、プログラムの先まで見えると、難しそうという先入観が生じてしまう。プログラミング言語は、英語がベースであることもその一因であるが、日本語プログラミング言語は、現状、教育用言語としても普及していない。教科書なしでのリアルタイム入力には、教員側にも能力と習熟度が必要である。しかし、それらが十分でない場合は、作るプログラムの難易度を下げれば、ある程度まではカバーできる。入力する際にバグをわざと作り、プログラムを実行してから修正することもある。それによって、間違い探しも面白いものであるということを示す。教員が意図しないバグを出したときは、バグがあって当然という主旨の発言をしながら、「こういう現象が起きたときは、こういう理由から、こういう種類のバグであることが推測できる」という解説をする。そして、実際に、バグをリアルタイムで見つけて修正する。バグがどうしても見つからない場合は、ちがう書き方で最初から書き直す。しかし、その頻度は、2, 3年に1回おきる程度である。

上記の3.では、教員のプログラム入力速度が速すぎることのないように、チェックをする。具体的には「速度が速いか？」という問いかけを、常にパワーポイントで表示しておく。そして、教員の入力スピードについていけない学生には、クリッカーを押させる。

上記の4.では、教員はプログラムを入力しながら、「なぜこのようなことを実現するのにこういう処理を使うか？」「なぜこのような処理が必要なのか？」の2点を中心に解説する。リアルタイムで行う理由は、緊張感を持たせたいのと、後でまとめて解説すると、意味がわからないまま入力することが多くなるからである。

上記の5.では、「①完璧に理解した」、「②ほぼわかった」、「③なんとなくわかった」、「④全然わからなかった」の4段階をパワーポイントに表示する。学生にクリッカーを押させることで①～④のいずれかを選択させる。学生のプログラムの理解度の自己評価に応じて、教員は次の行動を選択する。①と②が8割以上であれば、これ以上のプログラムの解説をせずに、次のプログ

ラムの題材に移る。③と④が合せて5割より多い場合は、6.に移行して、ほぼ全ての行にコメント文を入れ、5.のチェックを再び行う。チェックの結果、理解度の改善が見られないようならば、似たような題材をその場で考え、もう一度1.から繰り返す。

この教授法の独自性は、リアルタイムでプログラムを入力し、リアルタイムで次に組むプログラムを決めるというところにある。教員はPDCAサイクルをリアルタイムで回すことになる。学生は小さなプログラムを、素質のない学生はなんとなく、素質がある学生はほぼ完璧に理解しながら書くことになる。この教授法は、書いたゲームのプログラムが動けば、プログラミング言語やアルゴリズムを理解しようという努力をやめないという仮説に基づいたものである。学生の理解度をチェックして、解説の詳しさと次に書くプログラムの難易度を決めるため、学生が全く理解できないという状態が起きない。まとめると、学生の心の中で生じるポジティブな自己フィードバックが、モチベーションの源泉となることを狙う教授法である。

この教授法が行われるゼミナールの規模は、年度や学年によって異なり、学生数8～18名である。2年次～4年次までの週一回のゼミナールで、ほぼ全員が地力で簡単なゲームを作れるようになる。

ただし、この教授法には欠点もある。１つは教員の能力や習熟度がプログラムに対し相応に高くないと、全く授業が成立しないこと。もう１つは、シラバス通りの進度で進められない可能性が高いことである。前者は教員の努力でカバーできるが、後者は学生のレベルによるので対処できるとは限らない。したがって、この教授法をそのまま使用するのは、ゼミナール形式の授業に限るべきである。

8　まとめ

文部科学省の現行学習指導要領は「生きる力」である。「ゆとり」か「詰め込み」か、ではなく「生きる力」をはぐくむ教育とし、基礎的な知識や技能の習得と思考力、判断力、表現力の育成を強調している。その全ての要素を同時に育成できるのがプログラミングである。本論文では、これまでのプログラミング教育の問題点を整理した。そして、学習者のポジティブな感情を燃料とするプログラミング教授法を提案した。

プログラミング教育で、論理的思考力を養成できるという見解がある。確かにプログラムは、ロジックを組み合わせたものであり、その見解は間違いではないであろう。しかし、より後の人生で役に立つことは、問題解決の楽しさ、目標達成の喜び、快感を得られることではなかろうか。同様のことはプログラミング教育に限らず、すべての教育にいえるはずである。

参考文献

［Spolsky, 2005］Joel Spolsky, “The Perils of JavaSchools”, http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html, 2005

［Sussman, 1996］Harold Abelson and Gerald Jay Sussman, “Structure and Interpretation of Computer Pro-

grams”, MIT Electrical Engineering and Computer Science, 1996.
［Bornat, 2006 ］Saeed Dehnadi and Richard Bornat, “The camel has two humps”, School of Computing, Middlesex University, 2006
［Ghory, 2007］Imran Ghory, “Using FizzBuzz to Find Developers who Grok Coding”, http://imranontech.com/2007/01/24/using-fizzbuzz-to-find-developers-who-grok-coding/,2007
［Hilfinger, 2012］Paul Hilfinger, “Structure and Interpretation of Computer Programs”, http://inst.eecs.berkeley.edu/~cs61a/sp12/2012, 2012
［Resnick, 2007］Mitchel Resnick, “All I Really Need to Know（About Creative Thinking）I Learned（By Studying How Children Learn）in Kindergarten”, Presented at Creativity & Cognition conference, 2007
［文部科学省, 2008］文部科学省，“中学校学習指導要領”，2008
［日本経済再生本部, 2015］日本経済再生本部, “成長戦略（素案）”，産業競争力会議, 2013
［Code］http://hourofcode.com/us
［前橋和弥, 2004］前橋和弥, “センス・オブ・プログラミング”，技術評論社, 2004

（2015.10.9 受稿，2015.10.14 受理）