# 大学でのプログラミング教育

美馬 義亮[1,a)]

**概要**：情報系の大学におけるプログラミングの導入教育は、その後の様々な科目につながる基盤となる教育であるため、すべての学習者に確実に、プログラミングの概念と、基本的なプログラミングスキルを習得させることがその使命である。プログラミングの学びにおいては、条件判断と計算の方法をこえて、配列、繰り返し、関数呼び出しといった抽象的な概念を扱う段階で、学びを困難に感じる学習者が多くなる傾向が見られる。最近は IDE をもちいて、プログラミングの実習を行うことが多く、その中では、基本となる概念を十分理解していなくても、一見問題なく動作するプログラムを完成させることができる場合がある。このため、教える側が、学習の遅れに気づかない場合もある。教育実践を通して見出したこれらの問題と、それらを解決するための方策について述べる。

## 1. はじめに

最近、小学生にプログラミングの教育をおこなうことや、プログラミングを体験することにより、計算的思考が身につくといったことが謳われるなど、プログラミングを教育の場で広く用いることに対する関心が高まっている。

本論のテーマもプログラミングの教育であるが、対象が異なる。情報技術者となろうとしている人たちのなかには、なかなかプログラミングが上達しない人がいる。理科系の教育の道程には、程度の差こそはあれ、人によって、なかなか越すに越されぬ難所がある。ここではプログラミングの導入期の教育に関わってきた筆者の 10 年以上の経験を振り返りながら、プログラミング教育を効果的にする方法について述べる。

### 1.1 プログラミングの専門家の教育

私が教育の対象としてきたのは、主として情報系の大学の 1 年生である。かれらの多くは、プログラミングに興味を持っているが、それまでプログラミングをおこなった経験のあるものは少ない。しかしながら、将来は情報技術の専門家としての知識を身につけて社会で活躍することを前提にした教育を受けようとしている人たちである。

[1] 公立はこだて未来大学
116-2 Kamedanakano, Hakodate, Hokkaido 041–8655, Japan
[a)] mima@fun.ac.jp

プログラミングの導入教育としては、C, Java, など広く利用されるプログラミング言語と近い文法をもつ Processing を用いており、変数とその型、条件分岐、繰り返し、関数呼び出しとその定義を、履修者の全員が理解できるようになることが目標である。

### 1.2 目標達成にかかわる理解度の問題

実際にプログラミングを教えてみると、十分な理解度に達した学生の割合を多くすることは難しい。良質のプログラマを採用する側から見ても、同種の問題がある。プログラミングの能力を見分ける方法の一つとして、インターネットで共有されている有名な問題として、Jeff Atwood による「Fizz Buzz 問題」という問題が存在する。これは、

> 1 から順番に 1 つづつ大きくなる数 N を画面に表示するのだが、例外として、N が
> 15 の倍数のときは「Fizz Buzz」という文字列を、
> （15 の倍数ではない）3 の倍数のときは「Fizz」という文字列を、
> （15 の倍数ではない）5 の倍数のときは「Buzz」という文字列を、
> それぞれ出力するプログラムを作成せよ

というような問題である。ここで、注目すべきことは、この問題をを出した時に、情報系の大学を卒業した卒業生であっても正解率が低いことである。

試しに、これを解いてみると、想像したよりは手

間がかかるとは感じられたが、それにしても、それほど難しい問題というわけではない。しかし、このような「プログラミングの入門教育」を行うものの立場からは、どういうところに問題があるのかを明らかにしておかなければならない。

```
01: #include <stdio.h>
02:
03: int main() {
04:   int i, n;
05:
06:  n = 100;
07:  for (i = 1; i <= n; i++) {
08:    if (i % 15 == 0) {
09:      printf("Fizz Buzz\n");
10:    } else if (i % 3 == 0) {
11:      printf("Fizz\n");
12:    } else if (i % 5 == 0) {
13:      printf("Buzz\n");
14:    } else {
15:      printf("%d\n", i);
16:    }
17:   }
18:   return 0;
19: }
```

ここでは、この問題が難しいものとした場合、どのように難しいのかを考えてみる。C 言語の特徴的な部分である、標準入出力に関連した include 文の使用や関数 main() の存在などがある。初心者にとっては、この部分はテンプレートを用意して、機械的に追加するものとして扱われることが一般的であろう。

また、printf() の使い方を学ばなければならないが、どのような言語でも、文字の入出力に関する問題については同種の問題はあるものと考えられる。この問題については、現実的には多くの学習者は、多少のとまどいを感じながらも、克服できるようである。

力技に近い、そして、知識の問題として克服されている上記の部分を除くとすれば、中心的な部分は、3 行目から 17 行目までの 15 行となる。これらの中にある要素を順にリスト化してみると、

- int 型の変数や宣言法の理解
- for 文の理解
- 整数の大小比較
- 整数のインクリメント
- 整数の余りを求める計算の記述法
- if-else 文の理解
- printf() と文字定数の記述法

などであろう。それでは、これらの要素のどの部分が学習者にとって理解困難なのだろうか？

以後、2 節では、単元に分割した考察について、3 節では、プログラミング環境について、4 節では、教育的な配慮、工夫について考察をおこない、この種の問題の改善についての考察を行う。

## 2. 単元による難易度の違い

プログラミングで学ぶ単元・概念は多くの教科書に共通点がある。これらを列挙し、難易度を比較してみると。経験的には、

- 変数、型、算術計算（易）
- 条件式と条件分岐（易）
- 繰り返し（やや難）
- 配列（やや難）
- 関数（難）
- 再起的関数呼び出し（難）

のような難易度の差がある。これらの、間違いの原因を考えてみると以下のようになる。(なお、以下の記述は、プログラミングが不得意な傾向にある学習者に特養の傾向であり、必ずしもすべての学習者の傾向を示すものではない。)

### 2.1 変数とその型、算術計算

変数について、その使い方がなかなか納得できない人には、ほぼ出会うことはない。

数値の整数、浮動小数点数の区別については、起こりうるのは、1/2 という式を 0.5 という値をもつと勘違いするような事柄である。

概念的にわからないということはなくて、多くは、型の自動的な変換をわすれてしまうことによる間違いが起こる程度である。

ただ、この種の問題を間違えたとき、誤答をしたものは「単純なミス」ということが多い。初心者であっても、これらの変数の型の概念が割り算計算の方法の違いであることは理解できている。

### 2.2 条件式と条件分岐

この条件式の概念には、概念的な混乱が起こるようには見えない。起こるのは、>=と書くべきところを=>と書いてしまうといったミスである。ほかには、数学の記法と混乱して、==とかくべきところを= と書いてしまう、あるいは `a < x && x < b` とかくべきところを `a < x < b` と書いてしまう、というようなところにある。これらの問題は、2、3

回間違いに遭遇し、改善を促されると、改善がなされるようである。また、このような問題は、Scratchのようなプログラミング言語ではそもそも起こらない問題なのかもしれない。

### 2.3 繰り返し

for 文、while 文による繰り返しは、プログラミング学習の最初に存在する関門の一つのように思われる。旧来の手続き型の言語では、制御変数の変化を記述することになっているのだが、その記述を理解することには、心理的な負荷がかかるとおもわれる。

初心者が、for 文を用いるときには、面倒なことを理解しようとするのではなく、あまり考えずにプログラムを組んで、その後にパラメータを定義しようとするような操作をしている様子を見かけることもある。

また、初心者にとっては、繰り返しの実行を行う場合は、繰り返して記述することによって解決できる場合もあり、積極的に呼び出しに用いられることはない。

### 2.4 配列

配列に対する操作は、繰り返しによる操作とともに現れるものであるが、繰り返し操作を使わない学習者にとっては、意味のない概念となってしまっている。存在意義がわかるまでは、「そもそも面倒そうな概念」として、敬遠されていることも多い。

### 2.5 関数定義

関数の呼び出しは、printf() をはじめとして、無意識的にライブラリの呼び出しを行う必要性があるので学んでおり、安定的に利用もおこなっている。

しかし、関数の定義においては、事情が異なる。コードのコピーをおこなって、変数の調整を行えばほぼ同等の機能が得られる。関数の定義を覚えることに慣れない学習者は関数定義を敬遠する傾向にある。引数や返り値という概念が無視されることも頻繁に見受けられる。

## 3. プログラミングスタイル・環境について

プログラミングを学ぶ人の多くは、パーソナルコンピュータを所有し、IDE(Integrated Development Environment) をインストールし、インターネットでプログラムのサンプルを検索しながら、思い通りに動作するプログラムを作成しようとする。

### 3.1 IDE を用いたプログラミング

IDE には、補完機能、エラー訂正の機能、自動インデントの機能があり、様々な、「サンプルプログラム」を切り取り、埋め込み、パレメータを変えて、試行錯誤的な修正を加えたのち、実行させることができるようになる。

学習者の中には、自分が必要とする動作をするプログラムをインターネット上でみつけ、パラメータを調整して動作させるといったプログラミングスタイルをとるものがある。このようなスタイルは、作成しているプログラムの、完全な理解がなくとも求められた動作をおこなう方法を提供する。

エドガー・ダイクストラは、万年筆でプログラムを書きつづけていたというが、当然、頭のなかで、プログラミングの振る舞いを想像し、その振る舞いを実現するソースコードを書き下ろすということをしていたのであろう。その対極にあるプログラミングスタイルが、このような、プログラムのコピーとあたかも、愛玩動物を調教するような、細部のパラメータの調整によるものであろう。

### 3.2 「できること」と「わかること」の違い

ここで、我々は、何事かができるようになることと、対象となることが、説明できるようになることは異なっているということを、明確に理解しておかなければならない。

コピーと調整によるプログラミングは、「外延的」プログラミングとでも呼ぶべきものであり、作り手にとっては、構成的な手法で構築されたものではなく、そのプログラムの動作の細部に対する説明ができるものではないことが多い。

教育一般について言うならば、このような学び方は、間違いとは言いきれない。自転車にのったり、水泳をしたりするときには、どの筋肉をどのように用いるのかなどど、分析をしているわけではないし、語学を学ぶときには、文法を学ぶよりコミュニケーションがとれることを大切にしたほうが良さそうである。

だが、プログラミングをおこなうということは、最終的には要求された仕様通りのプログラムの動作を保証雨することを要求されるものであるから、完全な理解をおこない、十分な説明をおこなわなければならないため、「わかること」が必要になる。高校の教科で数学という科目を習得するために、問題に対する解決法のパターンを飲み込むように記憶してきた学習者には、その「わかること」が理解しにくいと考える。

### 3.3 数学の学びとの類似点

プログラミングは数学の学習と同じように基礎的な部分を「わかった」と思うまで突き詰めること、そうして理解したことを積み上げて行くことが必要である。学習者にはそのような、基本の理解とそれらに基づく論理的演繹操作の重要さを意識させる必要がある。

もう一つ、大切なことは、抽象化の考え方を身につけることである。数学という教科では、理解をすすめるために、抽象化の概念を身につけることが本質的であるが、プログラミングの初歩の段階では、抽象的な概念である、繰り返しや関数の呼び出しといった抽象概念を避けることも可能な場合が多い。

## 4. 教育の工夫

以上のように、プログラミングの教育においていくつか満たすべきことがあるが、これらを満たし、有効な教育を提供するについて述べる。

### 4.1 目的をはっきりさせたプログラミング

ここまで、あまり述べてこなかったことではあるが、プログラミングの文法だけを教えると、「なんのための学びかわからない」という声があがるので、最初に 40 数行からなるピンポンゲームのコードを、非常に小さなコードから、構成的に拡張する過程を体験させている [1]。

このなかで、変数、代入、演算、条件分岐、条件式、繰り返しの概念を順次学んでゆく構成をとっている。15 週後にはカレンダー機能をもち自分の余滴を記入できる小さなアプリケーションプログラムを作成させている。

使用言語は Processing としているが、これは、非常に小さな前提知識をもとにして、幅広い範囲に渡る対話的でビジュアルなプログラミングを、実現できるからである。キーボードに触れてから、何週間かのうちに、インタラクティブに反応するプログラムを自分でつくって、コントロールできることを感じられるのは学習動機を大きくしている。

### 4.2 学習者対する学習対象概念の明確化

プログラミング言語を学ぶにあたって、教師側が、教科書を示し、単元ごとの記述を示すだけでは、学習者にとってはプログラミングをおこなうための一里塚を示されただけにすぎず、何を学ぶべきなのか、何を学んだのか明確にならない場合もあるようである。

まず、基礎的な概念を意識し、できるだけ定着する試みとして、単元ごとに 3 問からなる基礎的な問題を出題することにした。ただ、語学学習などの知識を問う科目では功を奏する、「(厳密に論理を追うことをせず、記憶力におおくを委ねるような) おおまかな理解を重ねること」が、学習を行うことであるという「勘違い」を防ぐため、単元ごとに、全問を完答することを要求した。

- 内容を単元というスモールステップで区切った
- 単元の学習が確実に行われたことを確認した

このことにより、単元ごとに要求される学びの内容が明確になり、学習者も学習単元の理解に意識的になった。また、単位認定の資格をえる必要条件として、2 度の完答をもとめて、回答にあたっては、偶然に頼らぬような工夫をしている。

### 4.3 「わかってないこと」がわからないこと

先のほうで述べたことではあるが、学習者は誤答に対して、「単なるミス」であるとか「凡ミス」を理由に考えることが多い。しかしながら、それらの結論を出したケースの多くで、再度の「ミス」が発生する。

これらの多くは、誤答の直接的な原因を言語化して確認することがない場合である。このようになる理由は、問題を細部まで理解していないにもかかわらず、理解していると思い込んでいる場合が多い。聞いてみると、このような不完全とおもわれる学習をしている者の多くが「だいたいわかっている」というのが特徴的である。

このような場合、教える側は、学ぶべきことを言葉として明確にし、「(わかっていないことを) わかっている」と思い込んでいる学習者に口頭で説明を求め、本当にじゅぶんな理解ができているのかを確認する必要がある。

### 4.4 教員は学習者の「わからなさ」がわからない

多くの教科で同様のことが言えると考えるが、プログラミングという教科に関しても、教員になっているような人に聴いてみると、初学者の頃は「繰り返し、関数呼び出しなどは、すぐに理解できた」と言われることがほとんどである。

ここで、問題にしたいのは、そのような教員には、理解が不足している学習者に対面した時に、「なにがわからない」のか、もしくは、どこまでわかって、どこがわからないのかといったイメージがつかめない場合があるということである。

プログラミングのどこが難しいのかについて、日頃から思いを巡らせておくことは重要である。理

解に苦しみ、悩んでいる学習者がいた時は、どこでわからなくなっているのか一緒に考える必要があることを意識すべきだろう。

### 4.5 ペアで確認することの試み

最近の試みとして、プログラミング課題の進捗をペアを組んで確認することを始めている。一人で閉じた環境でプログラミングを行うのではなく、学んだはずのことを、言葉にして伝え合うことが具体的な学習の確認に役立つと考えるからである。

さらに、その先には、どちらか一人がキーボードに入力し、プログラムを書き、もう一人がそのプログラムを確認しながら作業をすすめるペアプログラミングという手法 [1] も有効であろうと考えられるが、初学者にふさわしい実施方法はまだ確定していない。

## 5. まとめと今後の課題

以上でのべたことを、まとめると以下のようになる。

- 動機付けのために、対話的なプログラム作成は有効である
- 学習者が「わかる」ということわかっていない場合があること
- 教える側が、学習者がどのように「わからない」のかわかっていない場合があること

さらに、これらの問題を解決するために作ったプログラムの内容を確認するという方法は、正確さが不十分な場合もある。そのため、プログラミングに関連した基礎的な概念を確実に獲得していることを確認する手法を併用したほうが良い。また、確認範囲は小さくわけてスモールステップですすめるのが良い。

教育するものにとっては、自分の教育が効果的なものとなって欲しいと念ずるような気持ちもあるが、心理療法においてもその成果の要因は、それほど多くはないという説 [2] もある。これによると学習成果の要因は、

- 教育外要因 40%
- 教育関係要因（受容・共感、思いやり、はげましなど学びてと教えての関係）30%
- 期待、希望、プラセボ要因（ピグマリオン効果と呼ばれるもの）15%
- モデルや技法要因 15%

とも言われている。実際のところ、大学時代にプログラミングに関して指導に苦労していた卒業生に、「新入社員研修で 2 週間でプログラミングがわかるようになった」と伝えられた経験もあり、学びの要因には、様々なものがあり、教育技法を考えるだけではなく、「プログラミングの技術習得に向かう気持ち」をどう持たせるかについても、深く考えなければならないと考えている。

### 参考文献

[1] 情報表現入門 – Processing Programming、美馬 義亮：未来大出版会 (2014).
[2] ペアプログラミング –エンジニアとしての指南書ローリーウィリアムズ, ロバートケスラー：ピアソンエデュケーション (2003).
[3] S・D・ミラー，B・L・ダンカン，M・A・ハブル：心理療法・その基礎なるもの –混迷から抜け出すための有効要因–，金剛出版 (2000).

## Q & A

Q: 質問、A: 回答、C: コメント

Q: プログラミングについては、学生が時間をかけて納得しながら、練習できるようにすることが大切だと考えますが工夫はされていますか？（久野）

A: そのため、反転授業的に授業の中で、プログラムをつくり実行するという状況をつくりました。人によって、授業で聞くことを好む人もいるし、作業の速さも異なるということもあるので、どのようなやり方が良いのかについては、まだ試行錯誤が必要だとかんがえています。時間をかけてもらうためには、興味を引きそうなピンポンゲームなどのアクションゲームを題材にするなどの試みを行っています。

Q：課題は、小さなステップにして与えるのが良いです。あまり大きな課題を与えると、学習者にとってそれを乗り越えるのが困難な場合もあると考えます。(久野)

A：ご指摘のとおりだと思います。

Q：for 文のような概念を理解できない人は、自分で時間をかけることもしてくれません。どのようにして抽象化された概念の獲得をしてもらうのが良いでしょうか。(山之上)

A：ステップごとにプログラムが実行されるとき、変数の代入の変化を見るなど、何がおこっているのかきちんとフォローする経験を育むのが一つの

方法ではないでしょうか。

Q：for 文が分かりにくいのは、初学者には「プログラムの記述が並行して実行されると勘違いされることが多いため」ではないでしょうか。初心者の中には、記述されたプログラムが、同時に実行されるような解釈をすることにより、動作の勘違いをしてしまう例もあると思います。プログラムは、一文づつ実行されることを、伝える工夫もしたほうがよいと思います。(中山)

A：確かに、プログラムでは、個々の文が順序に従って実行されていくことを明示すると学習効果が上がるケースもあると思います。

C: for 文を構文から、すなわち、示されたプログラムの説明から入るから学生が理解し難いのではないでしょうか？アルゴリズム、すなわち、繰り返しを展開した動きの具体例、変数の値の変化、意味を示してから、それを抽象化して繰り返しのアルゴリズムにまとめた上で、繰り返しの構文を教えた方が理解しやすいでしょう。モデル (意味) を示してから構文を教えるべきだと思います。(伊知地)

Q：高校数学の中にはプログラミング言語における変数のようなものは存在しないし、プログラミングカウンタのようなものも想像させにくくなっています。うまい方法はあるでしょうか？ (伊達)

A：たとえば、アセンブラをつかうと、そういった概念はずっとわかりやすいものになります。ただ、論理回路や電子工学のような低いレイヤの構造を追って学べばよいというものではなく、計算のモデルはある程度抽象化してわかりやすく提示するということも求められていると考えます。これも工夫してゆかねばならないことなのでしょう。

Q：「長く、プログラミングの理解が腑に落ちずにいた卒業生が、就職後 2 週間の研修でとてもよくわかるようになったと報告した」という話が紹介されたが、そこでは、本当に理解できていたと考えますか？（まだ、できるようにはなっていないのだけれど、「単に、できるようになったといっただけ」ということもあるので、そのようなケースではないかという確認でした。）(どなたか不明です)

A：本人の言葉は、かなり確信のもてたと思える言葉でした。また、実際にプログラミングの仕事をしていることも、話していたので、信憑性は高く、その人はプログラミングを理解できていたと確信しています。