何のためにプログラミングを学ぶの? そしてどのように? [1]

久野　靖†

電気通信大学情報理工学研究科

〒182-8585 東京都調布市調布ケ丘 1-5-1

y-kuno@uec.ac.jp

概要

現在、初等中等教育におけるプログラミング学習が注目を集めている。とくに小学校におけるプログラミングの必修化は今年になってから急に具体化し始めている。その一方で、高等教育を経てきた人たちでも（場合によっては情報系の出身者でさえ）プログラミングができない、わからない、という状況も普通に見られる。本発表では「そもそも何を目的としてプログラミングを学ぶのか？」「その目的に照らしてどのような学び方が望まれるのか」を整理して示した上で、関心を持つ参加者と議論をおこないたい。

1　はじめに

本稿は「夏のプログラミングシンポジウム 2016 教育・学習」の発表原稿として作成している。それに先立ち、1 月の冬のプログラミングシンポジウムにおいて、夜の自由討論で「情報教育」のセッションを長野大学の和田 勉先生と一緒にコーディネートさせて頂いたが、多くのご意見を頂けてそれなりに盛り上がったことから、上記のテーマ [2] で夏のシンポジウムを開催しようということになったものである (と、勝手に理解している)。それで、冬のシンポジウムでは情報教育全体についての自由討論だったが、自分では「プログラミングをなぜ学ぶか・どのように教えるか」がいちばん興味のあるところなので、本発表ではその部分に絞って取り上げ、多くの方に議論を頂きたいと考えた。

なぜプログラミングを学ぶか、というテーマは現在多くの人の関心を集め、多様な議論があるところである。筆者もたまたま機会があり、このテーマで解説 [2] を執筆したばかりなので、なぜという部分についてはこの解説で述べたことを再掲して紹介し、議論の題材としたい。

そして個人的には、「なぜ」については色々な意見があってもそれほど大きな問題はないが、[3]「どのように」学んでもらうかの方が喫緊の課題になっていると思うので、こちらを中心に考えていることをお話しし、また自分が担当しているプログラミング入門的な科目の経験・データを裏付けとして紹介したい。

2　なぜプログラミングを学ぶか

2.1　考えられる目的のリスト

なぜプログラミング学ぶか、すなわち「目的」については、上述のように多くの議論があるが、[2] ではそれらをなるべく広く収集し分類したので (書籍 [3] なども参考にした)、それを以下に再掲する。以下のリストは大まかに「職業的必要性 (V1〜3)」「教養 (L1〜4)」「表現力・創造力 (E1〜3)」「価値ある体験 (X1〜2)」に分類できると考えている。

ソフトウェア開発者が必要 (V1): 今日では世の中のあらゆるところでコンピュータが使われており、これら多数のシステムを動かすソフトウェアを誰かが書かなければならない。そのような職に就く人は当然、プログラミングを学ぶ必要がある。

仕事の一環でプログラミングが必要 (V2): コンピュータの広範な普及の結果、ソフトウェア開発者でなくても、自分の仕事のために必要なプログ

---

[1] Why should we learn programming? And how? by †Yasushi KUNO, Faculty of Information and Engineering, University of Electro-Communications, Chofu-city, Tokyo, 182-8585,JAPAN.

[2] 夏のシンポジウム 2016 全体では、人間の学びに限らず、機械学習なども含んだテーマとして発表募集している。

[3] もちろん、全く問題がないわけではなく、できるだけ多くの方に「なぜ」についてきちんと知っておいて欲しい、と考えていることは確かである。

ラムを作成することが増えてきている。[4] 今後ますます、プログラミングを学んでいなければ就けない職業が増えると見込まれる。

ソフトウェア技術者との連携のため (V3): ソフトウェアを発注するなどの形でソフトウェア技術者と連携する必要のある仕事は多い。そのような場合に、技術者と有効なコミュニケーションを取り、効果的な開発を行うには、プログラミングの経験や理解が有効だと考えられる。

コンピュータの原理理解のため (L1): コンピュータの本質が何であり、何ができて、なぜ今日のように広まったのかについて真に理解するには、自分でプログラムを書いて動かす体験が必要である。これは、既存のプログラムを使うだけでは「そのプログラムでできること」しかできないのに対し、自分でプログラムを書くことはある意味では自分がコンピュータになることであり「コンピュータができることは可能性として何でも作れる」からだと考える。

論理的思考/計算的思考を身につける (L2): コンピュータはプログラムの指示通りに動作するので、プログラムを書く際には起こり得るすべての場合を想定し、それぞれの場合について厳密に動作を記述する必要がある。このため、プログラミングを経験することは筋道立てて系統的に考える練習となる。またその記述に必要な特徴的な考え方は「計算的思考」(computational thinking) と呼ばれることもあるが、このような考え方を知っておくことはプログラム以外の場面でも役に立つ。

問題解決と能動学習の題材として (L3): コンピュータは強力なツールであり、多くの問題に「解答を導く」「人間に代わって作業してくれる」という両面から解決策を提供し得る。このため、プログラミングを通じて自分が持つ問題を解決することを (自分の問題としての～能動的な) 学びの題材としやすい。

答えが 1 つだけでない題材として (L4): わが国の初等中等教育は (とくに大学受験前後で)「想定された唯一の正解を当てる」ことに偏重し過ぎており、これが個人が社会に出て答えのない問題に取り組む際の妨げとなっている。プログラムは同じ動作を実現する複数の記述が可能であり、「解の多様性」を身をもって学べる題材である。

自己実現/表現の手段として (E1): 自分で考えてプログラムを作ることは、自分のアイデアを表現し形にすることであり、強力な表現手段である。そしてプログラムを完成させることは、自信や達成感をもたらしてくれる。

もの作りと創造力のため (E2): ソフトウェアは実際に動いて役立つものなので、プログラミングは (コスト・安全性・場所・機材・身体能力等に大きく制約されること無く) もの作りの活動が行える題材となり、創造力を育む機会を提供してくれる。

思考を外部化した成果物として (E3): プログラムは「自分が考えたこと」を厳密かつコンパクトに外部化したものであり、それを自分や他人が見て検討することは「個人が考えたことを論理性・客観性を持ってグループで検討する」経験となる。

楽しく熱中できる題材として (X1): 人間は楽しんで熱中するとき最もよく学ぶ。プログラミングはコンピュータに指示して思い通りに動かすという点で、多くの人にとり楽しく熱中できる題材であり、熱中して学ぶ貴重な体験を提供してくれる。

試行錯誤の経験を積む場として (X2): 実社会における課題には、少し考えただけでは解決せず、さまざまな可能性を試行錯誤して行く必要があるものも多い。プログラミングは、他人に気兼ねせずいくらでも失敗してみられる場として、貴重な試行錯誤の経験を与えてくれる。

2.2 どの目的が重要か?

前節では目的のリストを挙げたが、それではその中でどの目的が重要だろうか? 実際に学習プログラムなどを設計する際には、目的に応じて設計が異るだろうから、目的を絞り優先度を付けることは必要である。

筆者は前記のリストから、原理理解 (L1)、表現手段 (E1)、熱中できる題材 (X1) の 3 つがこの順でとりわけ重要だと考えている。その理由を以下に記す。

まず筆者は、わが国の社会では情報技術者が適正に評価されなかったり、適切な活躍の場を与えられないなどの問題がこれまで長く続いて来て、一向に改善されないという大きな問題があると考えている。そしてそのいちばんの原因は、世の中の「一般の人」は情報技術のことを知らなくてもかまわない、専門家に金を払って頼めばすむこと

[4] たとえばデータサイエンティストは、既存のソフトを使って分析するというより、自ら大量のデータを収集したり取り扱うためのコードを書くことが求められる職業だといえる。

だ、という空気が蔓延していることだと考えている。このような「あなたまかせ」の態度を許すということは、トップが情報技術者を適正に評価したり管理することを放棄することであり、また金を払う側がよい技術者と悪い技術者を見分けれなければ「悪貨が良貨を駆逐」するのが世の成行きである。このような問題を是正するには、大人を変えるのは困難だから、「学校教育において」「全員が」コンピュータのことをきちんと学ぶしか方法はない、したがって (L1) が重要、ということになる。

次にそのようにしてコンピュータの原理を皆が学ぶようになったとき、社会としての価値は上のようなことにあるが、学ぶ本人にとっての価値は「それによってこれまでにない多くのものが作り出せる」こと、すなわち (E1) にある、と考える。とくに、プログラミングによってどれだけ新たな表現が可能になるかは体験しないと分からないことであり、したがってこれも「学校教育において」「全員が」体験する価値があると考える。

最後の (X1) については、子どもたちが熱中できる題材はスポーツでも音楽でも沢山あるではないか、という反論を頂くことがある。もちろんそれらはそれぞれにすばらしい体験だと思うが、筆者は人間の最大の特徴はその「考える」能力にあると思っているので、その意味でプログラミングに取り組んで熱中するという体験が価値があると思っている。もちろんパズルやボードゲームや数学で考えてもよいが、現状ではこれらに熱中する子どもの数が多いというわけではないので、そこにプログラミングを加えることで考えることに熱中する子どもが増えて欲しいということである。

3 プログラミングの学び方は?

3.1 よくない学び方の特徴

いきなり「よくない」から話が始まって恐縮だが、筆者も筆者の同業者の方々も、これまでに (おもに大学生などが)「よくない」やり方でプログラミングを教えられて挫折してきているのを大量に見て来ているわけで、そこから「何がよくなかったか」を認識するのは重要である。

筆者の考えでは、その「よくない」要素はわが国の教育のやり方全般から来るものとプログラミングに固有のものとに大別される。まず前者から整理してみよう。従来の (そして今日の多くの) 学校の授業は次のようになっているものと思う。

- 教科書に書いてある沢山の知識をまず「そのまま覚える」ことを求められる。
- 教科書に載っているような練習問題は「どんな問題がある」「どうやって解く」をドリルなどで繰り返し練習させせられ覚えさせられる。
- 試験のときはその「覚えた」やり方で短時間に多くの問題を解くことを求められる。

しかし、このようなやり方はプログラミングを学ぶという点からは最悪である。先に挙げた目的群のなかで、プログラミングは自分独自のものを作り出す表現手段で、その新たなものをどのように作り出すかを考えることが楽しく熱中できる、という話になっていたのに、上記のやり方ではその「独自」「新しい」が否定されている。

さて次に、上記のような「従来型の」学習の延長にあるテキストについて特徴を挙げてみる。

- テキストにはプログラミング言語の規則 (書き方、機能) が逐一解説されていて、授業でもそれを順番に学んで行く。
- 演習問題が「プログラムを書くこと」でなくその説明している内容の知識を覚えたかどうかの確認問題になっている。
- プログラムの例題が少ししか登場せず、その少しの例題を懇切丁寧に説明している。

一番最初の点について補足すると、言語について一通り説明する、というのは本を執筆する著者にとっては分かりやすいフレームワークであり、そのためにこのような「本 (教科書)」ができやすいということあると思う (ここで自分はどんな本を書いてきたかなと思い返すと冷や汗が…)。

このようなテキストに沿った授業ではだいたい、プログラミング言語の構文や機能を逐一説明していって、その説明を暗記したかどうかの確認が時々おこなわれ、プログラムを書くのに必要な内容が揃ったところで例題に入るが、その例題も時間をかけて丁寧に説明しておしまいか、実習があるとしてもその例題をそのまま打ち込んだりすこし手直ししたりしておしまい、ということになりやすい。

しかしプログラムを組んだことがある人なら誰でも分かるように、プログラムは言語の知識を暗

記しただけでは書けるようにならないので、上のやり方だと実際には書けない学生が量産される。そして単位を落しまくるのではこまるので、試験問題はプログラムを書かせるのではなく、教科書通りの例題が再現できれば済むものになりやすい。となると、学生も例題の丸暗記で対応してくるので、結局単位は取れてもプログラムは書けない、という結果につながる。

こうして見ると、わが国でプログラミングの授業が失敗しやすいのは、知識伝達型の一斉学習が標準という学校文化の上に、解説書スタイルの教科書のできやすさ、それに素直に従った (あまり教育方法について考えていない?) 授業者が組み合わさるとこのような悲しい結末が待っているということになるだろうか。

以下ではこの「反面教師」パターンにならないためにはどうするか、という点から筆者が工夫してきたポイントを複数とりあげて説明する。

3.2 「離陸」とその維持

「離陸」とはもちろん、飛行機が地上滑走のあと空中に浮かぶことを指すことばだが、ここでは次の意味で使う。

離陸 — プログラムを書く人が「このようにしよう」と思ったときに、実際にそのようにプログラムを作れ、動かせること。

それは学び方ではなくプログラミング学習の最終目標ではないの、と思われるかも知れないが、そう思うこと自体がまさに「書けない学習者」を量産している、というのが筆者の考えである。

たとえば、飛行機の操縦を学ぶのも自動車の運転を学ぶのも、難しいところは教官に助けてもらいつつ、とにかく空中や練習コースで操縦/運転し、動かしながら細かいことを学んで行きますね? これがもし「あらゆる細かいことを全部教室で学び、それが全部終ってからさあ実習」だったら絶対に駄目だと思いませんか? ところが、上で説明してきたプログラミングの授業はまさにそれをやっているのである。

そこで筆者が提唱しているのは次のことである。

- 入門に使うような言語は 2〜3 行とかせいぜい数行で動くプログラムが記述できるはずなので、そのような例題を 1 つ選んで丁寧に説明し、どの部分が何を意味しているかきちんと納得してもらう。[5]
- 次に、その例題を打ち込んでそのまま動かし、動く様子を体験させる。
- 続いて、例題を自分の考えに応じて少し修正し (計算式の変更とかメッセージの変更など)、修正に対応した変化が動作に現れることを確認させる。

ここまでで「離陸」が完了すると考える。そしてこの、「自分で考えて作ったプログラムがその通りに動く」という体験はとても魅力的なのもので、(既に嫌な体験でスポイルされていなければ) 誰もが熱中できると筆者は考えている。

そして「離陸」が完了したらその後は、この状態を維持しつつ、少しずつ新しいことを追加して学ぶ、という形を取るのがよい。そのとき、新しいことはそれぞれ例題として確認した後、自分でもそれを使ったプログラムを作って動かしてみることで離陸を維持する。

このようにすれば、新しいことを学ぶつど、それによってこれまでにできなかったことが記述できる、自分の能力が増えたことが嬉しく感じられる。このため、新しい内容に次々に取り組んで行くモチベーションが得られる、というのが筆者の経験である。

これに対し、前節までのだめな方法では、机上で沢山学んだあとでようやく実習になるので、沢山のことを覚えていないとプログラムが書けないというプレッシャーがあり、また沢山学んだ後のプログラムは長く複雑なので入力で間違いを犯しやすく動かすのに苦労する。そうなると、教科書にある例題を正確にそのまま入力するのが最善というメンタルになってしまい、自分でプログラムを作るという意識から遠ざかってしまう。このような流れになったら、自力でプログラムが書けるようにならないのも当然である。

実際には、上のような問題に対処するため、最初の例題は平易で短くしている教科書も多くある。しかしそのような 2〜3 行の例題は見るからに簡単で易しいため、学習者も教員も「見るだけでいいや」と思ってしまい、スキップされがちである。そうやって「骨のある」例題から着手した

[5] Java のようにおまじないがくっついている言語は外側のおまじないは常に同じだから説明しない、ということでもよい。

ら上と同じことで、せっかくの教科書執筆者の温情も台無しである。

このような失敗を避けるためには、学習をリードする教員が「離陸とその維持」を最重要のこととして配慮し続けることが必要だというのが、筆者の考えである。

3.3 各自のレベルに合ったチャレンジ

授業など集団でプログラミングを学ぶ際の難しさとして、学習者ごとのレベルの違いがしばしば挙げられる。トレーニングなどでも個人の筋力に合った適切な負荷が必要であり、負荷が強すぎても弱すぎても能力向上に役立たない。同様にプログラミングでも、個人のレベルに合った問題で練習することが重要である。

これは本来、プログラムに限ったことではないのだが、プログラミングでは個人ごとのレベルの違いが非常に大きくなやすいため、とくに問題にされるのだと思う。

これに対する筆者の解答は、多様なレベルの練習問題を用意し、学習者に選んでもらう、というものである。どの項目であっても、その項目の新しい概念を説明する基本の例題があることは上に述べた。これに対して、さまざまな大きさのギャップを持つ練習問題を例えば次のように用意する。

- 計算する数値やその元となる数式が違っている
- 扱うデータの個数が違っている (2 個、3 個、4 個、N 個など)
- アルゴリズムの選択やアルゴリズム自体の工夫が必要である
- 学んだ概念をうまく使える別種の題材
- 学んだ概念をうまく使える学習者の好きな題材 (自由課題)

それぞれのレベルの問題でも、(自由課題を除いて) さらに細かい難しさの違いを設けたものを複数用意することで、多数の問題が用意できる。この中から学習者が適切なものを 2〜3 題選んで解くことで、そのセクションの演習ができる。

問題の中から学習者が適切なものを選ぶのは難しいのではと思われるかも知れないが、実際にやった経験ではそれはあまり問題なかった。というのは、学習者にとっては問題を呼んで「これはこうすればできる」とすぐ思い浮かぶ問題は易しすぎる問題であり、スキップしてもらえる。先頭から見ていってすぐには解答が思い付かないがあると、それは解答者にちょうどいい問題であるが、学習者の方も問題にチャレンジしてみたいと思うようで、それを選択してくれることが多い。

学習者による選択だと一番易しい (楽な) 問題ばかり選ぶのでは、という質問をいただくことがあるが、これについては「自分に合った問題をやることが自分が成長する機会になるのでベスト」という説得を繰り返すことでおおむね対応できると考える。

3.4 自分の問題を解くことを楽しむ

最初の方で述べたように、プログラミングの重要な利点は自分がやりたいと思うテーマに熱中して取り組む機会となることである。そのため、題材もできるだけ「自分がやりたいこと」すなわち自由課題であることが望ましい。前節の問題レベルの最後に自由課題があるのはそういう意味でもある。

しかし、まだあまりレベルのあがっていない学習者は「自分の好きな題材を」と言われても思い付かないことも多いし、思い付いた題材が自分の手に負えるかどうかも判断がつかない。

そこで筆者がよく使うのは「絵を描く」題材である。絵そのものは誰でも筆記用具などで描いた経験があり、「自分で描いてみたいもの」が具体的にイメージしやすい。さらに、幼児期から現在までにさまざまなレベルの絵を描いて来ているので、「現在の自分のプログラミング技能で作り出せる絵」という設定を考えることもあまり難しくない。そして、プログラムに間違いがあったときにその結果が絵の違いとして現れるので、プログラムの動作に対するイメージがつかみやすいという追加の利点もある。

上記と同様の考えによると思われるが、教育用の言語には LOGO のタートルグラフィクスをはじめ、絵を描く機能が充実しており、適切な言語/環境を選択することでどの学年でも扱うことができる。

絵でも自由課題でも、取り組んだ学習者の中には思ったことの全部は実現できなかった、という報告をする者も多くいる。筆者はそれもまた、必要なことだと考えている。というのは、どのようなレベルになっても「これ以上複雑なプログラムは作り切れない」という限界は必ずあるはずで、

そのことは早く知っておいてもいっこうに差し支えないからである。[6]

3.5 そのほかのポイント

以上が筆者がとくに重要と考えるポイントであるが、その他に必要と思うことを少しだけ補足しておく。

まず、プログラミングを学ぶというのは学習者が自分の問題に自分で取り組むことではじめて可能になるので (主体的な学び)、実習に入った後は教員の役割はファシリテータだ、ということである。演習中に質問されたことに答えるのは当然であるが、それ以外にも演習で詰まっている学習者を見るとつい横から教えたくなる。しかし、全部教えてしまうことは学習者の考える機会を奪う最低の行為なので、適切と思われるヒントを与えるにとどめるなどの配慮が必要である。[7]

もう 1 つ重要なのが、適切な言語・環境を選択することである。たとえば初等中等段階では、文字の読み取り十分できる以前であれば Viscuit のように文字を使わない環境がよいし、テキスト型であればキーボードによる文字入力を扱った後の小学校高学年以降がよい、英語のスペルが必要であればそれを学んだ後の中学生以上がよい、などのことが挙げられる。

大学での教育でも言語の選択はは当然重要である。たとえば現在でも C 言語でプログラミング入門を扱う大学があるが、C 言語は配列や引数のところでアドレスを理解してもらう必要があるので、最初の離陸のときに全部説明し切ることが大変という問題がある。そこで苦労するよりは、もっと簡単に取り組める言語で入門部分を通過し、どこかの段階で「新しい概念」として C 言語に取り組む方がいいのでは、というのが現在の筆者の考えである。

4 事例: 東大 1 年次科目「情報科学」

4.1 科目の概要

「情報科学」(2015 年度からは「アルゴリズム入門」に改称) は、東京大学の 1 年次の科目であり、

[6]もしかしたら思い付いたプログラムなら何でも書けるという天才もいるのかも知れませんが、筆者はそうではないので…

[7]筆者は以前はよく学生から「いま考えてますから言わないでください」と制止されていたが、最近は言われないので自分も少しは学習したかなと思っている。

#1	アルゴリズム, 変数と代入, 式の評価, メソッド定義, 浮動小数点と誤差
#2	制御構造, if 文, while 文, 数値積分, 計数ループ, 制御構造の組み合わせ
#3	if-elsif, 制御構造の組み合わせ (再), データ型, 配列の操作
#4	手続きと抽象化, グローバル変数, 再帰呼び出し, 中で枝分かれする再帰
#5	2 次元配列, 画像のデータ構造, 画像の生成, 様々な図形の塗りつぶし
#6	整列アルゴリズム, バブルソート, 単純選択法, 単純挿入法, ランダムデータによる時間計測, マージソート, クイックソート、時間計算量, ビンソート, 基数ソート
#7	時間計算量 (再), 既習アルゴリズムの別バージョン, 連立方程式の数値解法, 常微分方程式の数値解法
#8	オブジェクト指向, クラスとインスタンス, メソッド定義, 有理数クラス, 乱数, ランダムアルゴリズム, モンテカルロ法, 数あてゲーム
#9	動的データ構造, 単リストの操作, 抽象データ型, 式木, 動的分配, 抽象構文木の解釈実行
#10	スタックとキュー, 表と探索, 線形探索, 2 文探索木, 状態と状態遷移, 状態空間の探索
#11	動的計画法, 部屋割り問題, 2 次元の動的計画法, パターン認識, 遺伝子のアラインメント
#12	(様々な言語, 強い型, Java 入門)
#13	(復習, 前年度の試験問題解説など)

表 1: 情報科学 2014(久野クラス) 内容

理系ではクラス指定となっている (文系の学生は都合のよいクラスに登録できる)。科目の目的は、「情報科学の基本概念や思考方法をプログラミングを通して習得すること」となっていた (アルゴリズム入門では「アルゴリズムの基本概念や、アルゴリズムを作るための考え方を…」に変更されている)。また、言語としては Ruby が採用されている。

この科目は 2006 年に開始されたが、その際に東大の担当する先生がたが協力し扱うべき内容を検討し、最終的にそれらを集めた「標準スライド」なるものを作成された。これは基本的に、各授業の回に使うことを想定し、そこで取り上げる内容を説明するスライドの集まりである。内容を定めるものが標準スライドという時代はしばらく続いたが、その後教科書 [4] が出版され、現在はこの内容 (の指定範囲) を取り上げることになっている。

筆者はこの科目の開始時から非常勤として担当してきたが、「内容の範囲が同じであればよく、進め方は各担当に任せる」という方針だったので、

最初から自分がよいと思う順番に入れ換えて実施してきた。その方針は次の通りである。

- プログラミングの自然な習得に適した順とし、言語の機能を最初は最低限、次第に多く使うように並べる。
- なるべく早い段階で「絵の課題 (前述)」に取り組んでもらい、自力でプログラムを作る楽しさを体験してもらう。
- 言語の機能をひととおり学んで使いこなせるようになったら、情報科学の指定内容の落ち穂拾いをやって終わる。
- 全 13 回のうち＃ 1～＃ 11 は当日に 1 個プログラムを作成しレポートとして提出する「A 課題」を課し、さらに＃ 1～＃ 8 は次回授業までに 2 個プログラムを作成し提出する「B 課題」を課している。
- 各レポートは× (未提出)/△ (遅刻・不備)/○ (平常)/◎ (優秀) で評価し、要件を満たしていれば○とした。考察がない場合は△ (再提出可) とした。
- 成績は 19 個のレポートすべて○で 50 点 (◎があれば上積み) 換算、試験の満点を 50 点換算の合計とし最初からそのことを告知

これらの方針は、毎回簡単でよいからかならず自力でプログラムを書いてもらうことを意図して、このようにしてきた。

また、2011 年度からペアプログラミングを採用している。これは、毎回の課題について「2 人で」協力してプログラム作ることを可能 (推奨) としているもので、プログラムは同一でもレポートは別個に提出してもらっている。この場合、自分で書けなくてもレポートは出せるが、まったく分からないと考察が書けないので結局易しい問題を選んで自分で書くかペアの書いたものを十分教わる必要がある。なお、個人で作業したければそれでもよいようにしている。結果として、最初の方ではペアが多いが、回が進んでプログラミングになれてくると個人作業を選ぶことが多い傾向がある。

このように科目のもとの目的ぶっちぎりでプログラミング学習優先だが、実際にやってみると各回の「新たに学ぶこと」のテーマに標準スライドのどれかのお題をあてているので、最終的にはきちんと試験範囲をカバーして終っている。表 1 に本稿で分析をおこなった 2014 年度 (「情報科学」名称での最後の年度) の内容構成を示す。また、初期の資料をまとめて出版させて頂くこともできた。[1]

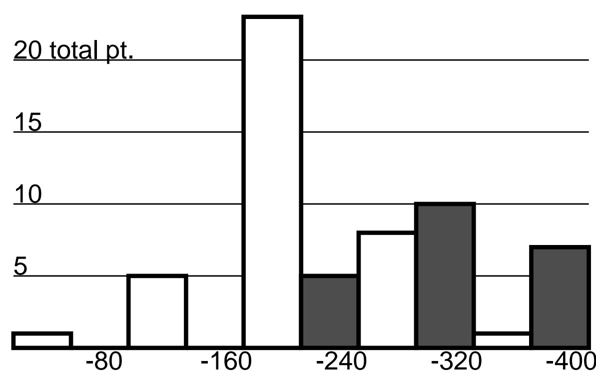

図 1: 総合点の度数分布 (gray:自由課題有)

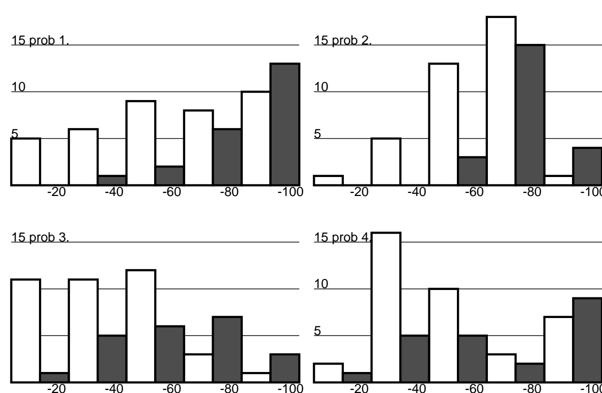

図 2: 大問ごとの度数分布 (gray:自由課題有)

4.2　プログラムが書けると試験ができるか?

さて、本稿のテーマは「プログラムが書けるとどういういいことがあるか」なので、共通試験の点数 (これは前述のように情報科学の基本概念や思考方法を習得しているかを見ている建前である) とプログラムができるかどうかの関係を調べることとした。対象としたのは 2014 年度の授業で「最後まですべての課題を提出し、試験を受けた」受講者 60 名のデータである。

まず分析対象とした 2014 年度の試験問題の各大問 (各 100 点、合計 400 点) の概要を示す。

問 1: ベクトルの加算や内積を (1 次元配列をベクトルと見立てて) おこなうメソッドやそれを下請けに行列積を計算する (再帰版の) プログラムを完成させる穴埋め問題。

問 2: 一定の勝ち負け率の勝負を反復して行うと

表 2: 試験点数のサマリー

	問 1	問 2	問 3	問 4	総合
最低点	0.0	10.0	0.0	0.0	20.0
最高点	100.0	100.0	100.0	100.0	370.0
平均点	69.4	63.8	44.7	58.4	236.3

きの N 試行後の持ち金を動的計画方で計算するプログラムを完成させる穴埋め問題。

問 3: 浮動小数点の誤差に関する 1 行程度の記述で解答する問題およびモンテカルロ法で面積計算を行うプログラムの実行結果を問う問題

問 4: 込み入った状況設定で起こる事象に従うお金の収支を動的計画法で求めるプログラムの穴埋め問題および白紙からプログラムを書く問題

各問および総合点の最大･最小･平均を表 2 に示す。

次に各回答者がプログラムをどれくらい書けるかについては、提出されたレポートのうち＃8B(時間を掛けてプログラムを作成する最後の回) をチェックし、次のデータを抽出した (本当はもっと色々求めたかったが時間が足りなかった)。

- 選択したプログラム課題の中に「自由課題」が含まれていたかどうか
- レポートに含まれていたプログラムの行数

まず、自由課題の無い受講生とある受講生に分けて大問ごと (図 2) および総合点 (図 1) のヒストグラムを作成した。総合点を先にみると、「無」は 161-240 点、「有」は 241-320 点にピークがあり、2 つの群がはっきり分かれているように見える。

さらに大問ごとに見ると、問 1 は配列と再帰まで分かっていればできる易しい穴埋めなので「有」は半分以上がほとんどできていたが、「無」はこれでもばらついている。問 2 は動的計画法だが取り上げた回が試験の日に近く試験に出やすいという予告もしたので「無」でもそれなりにできる人がいた。一方問 3 は何回か機会を見て取り上げている誤差の問題があまりできがよくなかった。最後の問 4 は複雑な問題であり、「有」の群でもかなり成績はばらついている。

次に、レポートに含まれていたプログラムの行数と試験点の散布図を総合点 (図 3) および各大問ごと (図 4) に作成した (縦横軸とも平均のところ

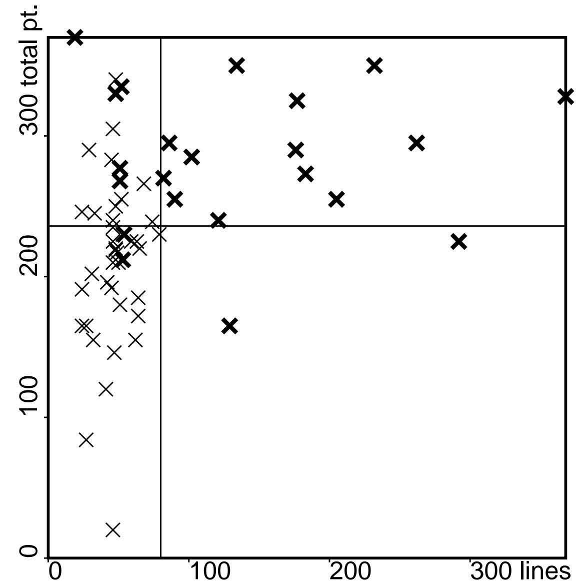

図 3: プログラム行数と総合点のプロット (太線: 自由課題有)

に直線を引いている。また太い×は自由課題「有」を示す)。総合点の方から見るとプログラム行数が多い方が得点は高いが、行数が多くなくても「有」の群はおおむね平均よりよい点数を取っている。

さらに細かく大問ごとに見ると、易しい/穴埋め型である問 1、問 2 は「無」や行数が少なくてもそれなりに得点を取った人がいるが、記述式を含む問 3 や穴埋めでないプログラム書きを含む問 4 では高得点は「有」やプログラム行数の多い人が中心となっている。

分析に用いたデータがレポート＃ 8b の自由課題選択有無とプログラム行数だけというのは明らかに不十分ではあるが、これだけで見ても「プログラムが書けている人は情報科学の試験もよくできている」ことは明確だと考える。

5 この先目指したいこと

筆者は現在電通大で 1 年次情報教育のデザイン・運営を担当する職にあり、2017 年度からの科目内容のリフォームに取り掛かっている。プログラミング入門科目 (「基礎プログラミングおよび演習」) は秋学期なのでこれから状況を見せていただくわけだが、昨年度までの状況について複数の先生がたからうかがった範囲では、プログラムが書けるようにならない学生が多くいるという報告もいただいている。

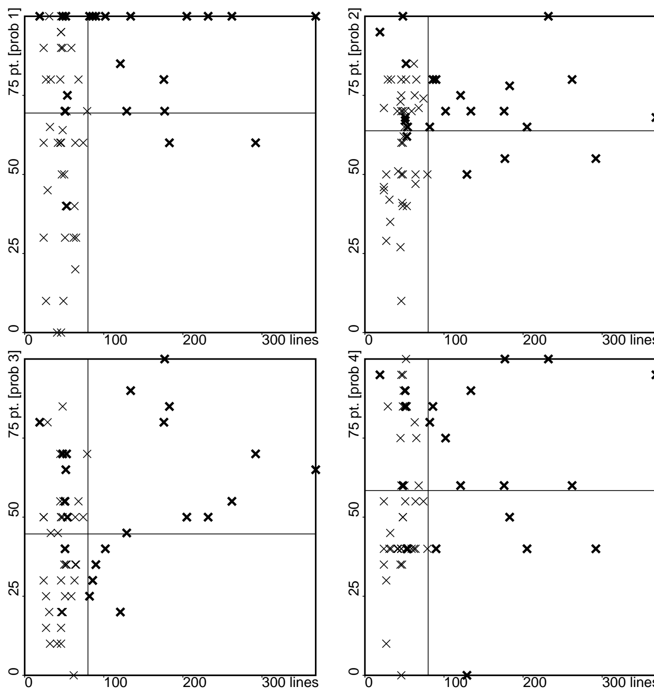

図 4: プログラム行数と各大問点のプロット (太線:自由課題有)

実際にどうするかはこれから検討するが、本稿で挙げたやり方を取り入れることで学生にとってより幸せな科目内容にできるとよいと思う。(本当はそのリフォームの前に本研究がきちんと完結して、それに基づいて進められるべきなわけですが、残念ながらそのようなわけには行かなそうです。)

参考文献

[1] 久野　靖, Rubyによる情報科学入門, 近代科学社, 2008.

[2] 久野　靖, プログラミング教育/学習の理念・特質・目標, 情報処理, vol. 57, no. 4, pp. 340-343, 2016.

[3] 神谷加代著, 竹林 暁監修, 子どもにプログラミングを学ばせる 6 つの理由, インプレス, 2015.

[4] 増原英彦, 東京大学情報教育連絡会, 情報科学入門 — Rubyを使って学ぶ, 東京大学出版会, 2010.

[5] 東京大学教養学部 情報図形科学部会, アルゴリズム入門 共通資料, 2015.

http://lecture.ecc.u-tokyo.ac.jp/johzu/joho-kagaku/

質疑のメモ

Q. 同じ科目の非常勤をやっているので補足します。2015 年度から再帰データ構造、オブジェクト指向は学ぶ範囲から外れました。アルゴリズム、数値計算の誤差、パターン認識をプログラミングを通して学びますが、1 年生の授業としてはパターン認識を学ぶのが特徴です。プログラミングを教えるのが目的ではありません。(伊知地)

A. 補足ありがとうございます。

Q. 「自由課題」とは具体的にはどのような課題? (渡辺)

A. 各回のテーマは「乱数」とか「動的データ構造」とか決まっているわけだけれど、「そのテーマを活用した、自分の作りたいプログラムを作れ」というのが自由課題です。

Q. 「自由に作っていい」と言われても困ってしまう学生が多いのでは? (谷)

A. もちろんそうなので、各回の課題は「データを 2 個から 3 個とか N 個に増やす」のようなギャップの小さい課題から高度な課題まで並んでいて、その最後に自由課題がある。学生はその中から好きなものを選ぶ。だから思い付かない学生は選ばないでよいということ。

Q. 自由課題のようなものが、最初の回からできるのか? (美馬)

A. 確かに最初の方はやりづらい。3〜4 回目くらいからやる。配列やったら「配列使って何か好きなものを作れ」とか。

Q. 「人工知能」の講義で Prolog を扱っているのだけれど、プログラミングの説明自体には、せいぜい数回のコマしか割当てられなくて、離陸した段階で終わってしまう。どうしたらよいか?(飯尾)

A. この方法を適用できると思う。Prolog なら 3 行とかで面白い動作ができるのだから、それを十分反芻してマスターしたら、バリエーションを考えて作れる。それが離陸。それからその状態で新しい概念を増やしていけばよいと思う。

Q. 自分は 3 年次まで前後期みっちりやる課程の 2 年次の「アルゴリズムとデータ構造」で御説でいう「ダメなやり方」をやっているのだが? (伊達)

A. すみません (笑)。新しいデータ構造が出て来るごとにそれをマスターしたらできるワザを楽しむとかできるのではと思う。ただ、1 年次の授業で離陸させていないならこの手法の前提が成り立たないわけで。

A'. 東大の授業「情報科学」は 2014 年度まで 1 回 90 分、2015 年度から「アルゴリズム入門」の名称となり 1 回 105 分で、毎回、講義と演習が半々です。(補足)(伊知地)

Q. まったくできない人にはどのように対処するのか? (美馬)

A. いちばん易しい課題は本当に易しいのでできないということはないのだが、ずっと易しい課題ばかりで行く人は面白くなれないまま来なくなるかも。自分に合ったレベルの問題でないと成長できないし面白くもないですよ、もったいないですよ、と説得はしている。

Q. この授業は必修か選択か? (谷)

A. クラス指定という準必修だが、必要だと思ってもらえないのかハードだと勘違いされているのか取ってくれる比率は半分より少ないくらい。必修になるとまた状況が違うだろう (チャレンジが増えるだろう) とは思っている。