「プログラミング学習の指針」の有効範囲

久野　靖[1,a)]

概要：小学生によるプログラミング学習が広まりつつあり、その進め方も多様なものとなっている。小学生対象の場合、つまづきが起きにくいようなプログラミング言語・環境を採用することが多いが、一方で伝統的なテキスト型言語でプログラムを作れるようになるというニーズもある。筆者は大学初年次のプログラミング入門科目設計を担当しており、その進め方の指針を取りまとめて来た。この指針は少なくとも大学生には有効だと考えているが、もしかしたらもっと若い学習者、そしてもしかしたら小学生にも有効かも知れない。本稿ではこの指針を小学生むけにアレンジした学習の進め方を検討する。

キーワード：プログラミング学習, テキスト型言語, 科目の設計方針

1. はじめに

「プログラミングを学ぶ」ということは今日、ポピュラーな学習の目標の 1 つである。少し前までは、プログラミングの技能は専門家だけが持つものであり、一般の人には関わりのないものである、という雰囲気が世の中にあったが、今ではそのような考え方は少数派になりつつある。その理由としては、次のことが挙げられそうである。

- 「アプリ」ブームなどを経て、プログラムは小中学生でも作って公開でき、他の人に使ってもらえるようなものだということが知られるようになった。
- プログラミング体験イベントや常設のプログラミングスクールが増え、既存のさまざまな「習い事」と同様にプログラミングが学べるものだという認識が広まった。
- 中学 (技術科)・高校 (情報科) のカリキュラムの改訂により、多くの生徒が授業の一貫としてプログラムを体験するようになった。
- 政府の「AI 戦略」などで、AI・データサイエンス人材が求められる (従ってこれらを学ぶ価値がある) と周知されるようになった。

そして 2020 年度からは、小学校の学習内容にプログラミングが取り入れられ、全員が学ぶこととなっている [*1]。ただ、2020 年度からの小学校プログラミング教育は筆者にとっては「やりたいことと違う」感がある。それは次の 2 点による。

- 学習の目標はあくまでも「それぞれの教科の内容」であり、プログラミングはその内容を学ぶ「手段」として取り入れられている。
- プログラミングを取り入れることの目的は「プログラミング的思考」「論理的思考力」を身につけること、となっている。

前者については、小学校にはまだ「情報」という教科がなく、各教科の中にプログラミングの内容を加えたことから来ている。そして後者につい

[1] 電気通信大学
[a)] y-kuno@uec.ac.jp

[*1] 学年進行ではなく、2020 年度の小学校 6 年生は、その 1 年間でプログラムの内容を学ぶ必要がある。

ては、何を目的とするか、ということを定めるにあたって、単に「プログラミングができるようになる」とは (反発がある、「できるように」なることの難しさ、裏付けとなる理由がつけにくいなどのことから) 書けなかったためと想像される。

これらの公式な扱いはさておき、筆者は、プログラムを学ぶことの価値は色々あるが、「自分でやってみたいと思うコンピュータの動作を自在に作り出せること」(長いので以下では「離陸」と呼ぶ) がその最大のものであり、従ってぜひともそれができるようなプログラミング学習を目指したいと考えている

その一方で、「離陸」という目標は「単にプログラミングがどういうものであるか体験する」ことよりはレベルの高い目標であり、その実現にはいくつもの工夫が必要である。

筆者は所属大学 (理工系単科大学) で初年次プログラミング入門科目の設計・運営を担当しており、その目標は「卒業研究などに進んだときに、研究の手段としてプログラミングの技能が役立てられる」ところに置かれている。これもまさに「離陸」が目標と言ってよい。

その目標を達成するため、いくつかの指針を考え、その指針に沿って科目設計を進めることで、一定の成果は得ているものと考えている。[7][8] これらの指針や進め方を小学校の生徒むけに「流用」することは可能か、また流用するとしたらどのような改造が必要となりそうかを検討することが本稿の内容である。

以下 2 章では、ひとくちに「やってみたいものを実現」といっても複数のレベルがあることを指摘し、整理する。3 章では筆者が大学初年次プログラミング教育の設計において採用している指針について紹介する。4 章ではこれを改訂して小学校の生徒むけにすることを検討する。5 章ではこの指針に基づきテキストを作った経験を報告し、6 章で議論とまとめをおこなう。

2. 「離陸」とそのレベル

上では「離陸」を「思うような動作が作り出せる」と定めたが、少し前に書いたもの [5] では、「自分の手でコードを書いて動かし、結果を見て手直しできる」としていた。自分で書けて手直しできなければ、思う動作に向かって作ることはできないので、これらは本質的には同じであるが、今の定義の方が少し抽象度の高い言い方になっている。

また、これらを見比べるうちに、ひとくちに「離陸」といっても、カリキュラムの組み立て方や教材の構造に応じて、次のようなレベルがあるのではと考えるようになった。

L0 (固定目標・固定経路) — 作りたい「目標」が予め決まっていて、そこに至る経路も決まっている。内容の決まった組み立て理科教材や Hour of code のように、実現するゴールは与えられており、それに向かって決まった作り方で組み立てる。経路は厳密に 1 つではなくても、Hour of code のように「想定正解 (バリエーション含む)」から外れて作ることが難しい場合は経路は決まっていると言える。

L1 (固定目標・自由経路) — 「目標」は与えられているが、そこに至る経路には自由度があり、経路を工夫することが求められる。アルゴロジックのように「旗を取る」目標は決まっているが、どのように動きを組み合わせて取るかは大きな自由度があるものがこれに相当する。

L2 (箱庭・砂場) — 「目標」のドメインが「画面に絵を描く、動かす、音を出す」など限定されており、それを実現する汎用的な材料が提供されている。Viscuit、Dolittle、Scratch、Processing など多くの教育用プログラミング環境はこれに相当する [*2]。また、汎用言語でも特定ドメインに限ってカリキュラム、教材を用意する場合はこれに相当する。

L3 (汎用言語) — 目的を限定しない汎用プログラミング言語で自由に目標を設定しておこなう場合。目標に応じてライブラリや言語を選択することまで含む。大学の研究レベルでプログラミングを扱う場合この水準が目標となる。

L4 (質的に異なる目標の実現) — 汎用言語レベル

[*2] これらの言語も汎用言語と同等の能力は備えるため、ライブラリの使用などもあり L4 としての使用も考えられる。

では実現が繁雑となる新たな (質的に異なる) 目標を設定し、それに呼応してプログラミング言語やパラダイムから作り出す。プログラミング言語研究者が行うような活動に相当。

なお、L0 が一番最低というわけではなく、その下に「言われた通り操作したりコードを打つだけで、目標は何もない」という「マイナスの」レベルがあり、実は現存するプログラミングの授業の多くがそのレベルだと考えられる。そのようなレベルでは学習者は学習活動に意味を見出しにくい。その意味では L0 も十分価値があり有用な内容だといえる。

これらを総合して見た場合、L0 と L1 は学習者が想定から逸脱しないので授業者にとって扱いやすい。また与えられる「目標」を学習者が自分の目標として受け入れられるなら、高いモチベーションや楽しい体験につながるが、そのことは必ずしも保証されない。プログラミングの学習活動として見た場合、目標が固定なので、「いつまでも」このレベルに留まることは無理がある。

これに対し L2 は、ある程度広く学習者にとって興味が持てそうな範囲を目標のドメインとするため、多くの学習者が意味を持つ (自分のものとして受け入れられる) 目標を見出しやすい。ただしそのためには適切な授業設計やファシリテーションが必要となる。目標ドメインも経路 (言語メカニズム) も汎用性があるため、長い時間とどまって発達し続けることができる。

L3 が L2 と異なるのは、もともと「さまざまなものを作り出す」ことが汎用言語の目的であるため、そのための材料が揃いやすいことである。特定の環境や条件 (例: ゲーム機上でネイティブコードで動く) を求める場合、これしか選択肢がなくなることも多い。L2 と比べて「学びやすさ」「つまづきにくさ」は最優先でないので、学ぶときの障害は多くなりやすい。

L4 はそのようなものまで考えられるという意味で挙げたが、本稿では扱わない。ただし、コンピュータサイエンスの専門家を育てる、という文脈では考える必要があると思われる。

大学のプログラミング教育は L3 前提なので、次章で述べる方針もそうなっている。小学校で扱う場合の様々な知見は L2 のものが中心である (それより低いものやマイナスのものももちろんある)。そこで、L2 による実践の知見を参考に「L3 を小学校 (おそらく高学年) に適用できるように指針を改訂する」というのが本稿のテーマである。

3. プログラミング学習の指針

3.1 避けるべきスタイル

前述のように、筆者は所属大学におけるプログラミング入門科目 (1 年次後期、全学必修) の設計・運営を担当している。その設計に際して、過去のプログラミングの授業や教科書を振り返り、このようなやりかたは「よくない」、というものをまとめたものを 2016 年の夏のプログラミングシンポジウムで発表した [6]。その部分は今でも変わっていないので、抜粋して紹介する。

「よくない」要素のいくつかは、次のような、わが国の教育のスタイル全般に由来している。

- 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。

プログラミングは自分独自のものを作り出す手段であり、その点が最も面白いのに、上のやり方ではその「独自」「新しい」が否定されることになる。次に、このような「従来型の」学習の延長にあるテキストについて考えてみる。

- 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。

1 番目について補足すると、言語について一通り説明することは、テキストの執筆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テキストが多く生み出される。それら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、解説書スタイルのテキストのできやすさ、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った結果であると思う。

3.2 採用した指針

前節の「避けるべきスタイル」を参考に、科目設計の指針をまとめた。詳細は [7] にあるが、ここでは指針と簡単な説明を挙げる。

P1: 離陸ファースト — 離陸については既に繰り返し述べたが、離陸ファーストとは「まず最初に離陸し、それを維持しながら個々の内容を学んで行く」プロセスを指す (図 1 上側の経路)。その利点は、常に離陸した状態で練習することにより、練習が楽しく、また自発的な学びが期待できることである。これに対し、前節のスタイルでは知識の習得を先に行うため、その後で順調に離陸できるかどうかは個人の資質や努力等に依存してしまう (図 1 下側の経路)。

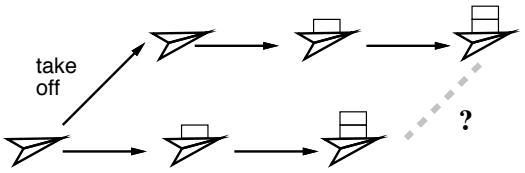


図 1 離陸ファースト

初回からの離陸ファーストを実現するためには、初回で扱う題材は十分少なく、なおかつそれだけできちんと動かせる内容に精選し、その上でそれらを自分の「道具」として使いこなせるまで試すことを促す。

P2: 多様な水準の演習問題 — 新たに学んだことを「道具」として使えるようになるためには練習が不可欠であるが、その練習は「適切な負荷」である必要がある。負荷が軽すぎても重すぎても意味がない (図 2)。

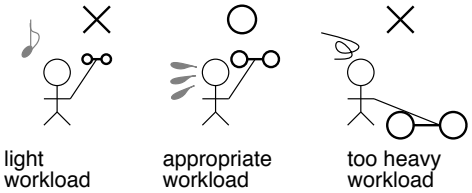


図 2 適切な負荷の必要性

たとえば個人教授で熟知した教授者であれば、学習者の水準を見極めて適切な演習を課すことは可能だろうが、大学の授業などでは難しい。そこで、演習問題を易しいものから難しいものまで多数用意し、学習者が選択するようにする。これにより、各々の学習者が自分に合った問題で適切に学べるようにする。また、実施に際しては、「難しい問題をやったらより偉くて点数が稼げる」という思い込みを打破する設計が必要である。

P3: 演習の重視 — 離陸ファーストを採用すると、最初の (ないし冒頭 2〜3 回の) 内容は多くの学習者にとって極めて易しいように思えるはずである (学ぶ内容を精選するため)。そのような場合、よく起こることは次のことである。

学生は時点 X までは「易しいと馬鹿にして」演習せず、時点 X で突然「難しすぎて落ちこぼれた」と気づく。

実際には、例題を写していると簡単に見えても、ゼロから自分でプログラムを書くとなると急に手が止まり、何も分からない気がして難しく思えるものである。そこで、初回から毎回、十分に演習に時間を掛け、この「難しさ」を体験し克服してもらうことで、離陸を維持し続ける。

P4: 苦手な人の学びを促す — 大学の初年次必修科目としてプログラミングを実施する場合、既に経験してきて得意な少数の学生と、何も知らない多数の学生が混ざった状態で実施することになる。そこで形成的評価 (授業時の評価や期中レポートの評価など) を絶対評価すると、得意な学生は楽によい点を取り、多くの学生はそれを見て否定的な感情を持つ。それではその先の学習がうまくいかなくて当然である。

これを避けるために、相当無理にでも形成的評価では得意な学生と一般の学生の差がつかないようにする必要がある。筆者は課題を「プログラムを説明するレポート」とすることでプログラム自体による差をつかなくした上、「S」をつける比率を数%と少なくしている。これはそこそこできる学生には気の毒であるが、期末試験で普通に絶対評価することで対応し、期中には我慢してもらう。

P5: 「プログラムが書ける」目標の明示 — プログラミング言語には膨大な規則が伴っているので、プログラミングを学ぶことで、それらの知識も併せて身に付けることになる。そのため、「知識を覚えてそれを試験で評価してもらう」という慣れ親しんだモデルに従おうとする学生が多く現れる。しかし実際には、規則に注力することは前節で挙げた避けるべきスタイルに相当する。

そこで学生が知識に注力するのをできるだけ避けるために、授業期間中の小テストや期末テストをすべて「プログラムを書く」テストとし、プログラムが書けること「だけ」が高い評価につながるように科目を設計した上で、そのことを予め宣言するようにしている。また、知識についてはテキスト PDF ファイルをすべて試験中に参照できるように提供することで「覚えなくてもよい」メッセージを出している。[*3]

ここで問題なのが、プログラム作成問題の採点である。一般にプログラムを記述させる問題を出題した場合、その採点は経験者が人手でおこなうためコストが高い (自動採点はプログラミングコンテストのように「完全正解か否か」しか出力しない場合には可能だが、部分点は付与できないので、小テストや期末テストには向いていない)。筆者らは情報入試研究会で開発した「短冊型テスト [4]」[*4] を採用し、正解との編集距離に基づき部分点を付与する自動採点により、小テストと期末テストの人手による採点を不要としている。

P6: 書き方の自由度の明示 — 今日の日本の教育は「問題には唯一の正解がありそこに早く到達すると勝ち」という強い刷り込みを作っている。これは社会に出てからさまざまな問題を解決する上でマイナスであるし、プログラミングの科目においても「テキストの例題の丸暗記」「誰かの正解のコピー」のような無意味な行為につながっている。

これを避けるためには、早い段階から繰り返し「コードには複数の書き方が可能」「どれか 1 つが正解ということはない」「自分の中に書き方の基準を作る」などのことを提示している。また、短冊型テストでも積極的に複数の正解が可能なように選択肢を作成している。

P7: プログラムは自分の頭で作り出す —

プログラミングの科目には、テキストにある例題プログラムがそのまま再現できれば単位がとれるようなものもある。しかしそのようなスキルをいくら培っても、実際にプログラミングが必要になったときに「まだ無い (必要な) プログラムを作り出す」ことができなければ意味がない。

このため、ミクロレベルでは前項の「自分オリジナルの書き方」を推進するが、マクロレベルで

[*3] 試験中にテキストを熟読していたらすぐ時間がなくなるので、内容を一通り身に付けておく必要はある。細かい事項まで記憶することは不要にしているということである。

[*4] 正解プログラムを行単位でバラバラにして重複行の除去・惑わし行の追加をおこなった上で解答記号を付し、回答者には行を選択して並べて正しいプログラムを組み立てることを求める方式。

は課題として「自分が作り出したい絵や動画を作る」など、本質的に自分固有の内容になるような演習を取り入れている。「配列」など個別の学習事項においても、選択可能な演習課題の中に「配列を用いて自分が作りたいと思うプログラムを作る」のような自由課題を含めるようにしている。

4. 小学校向けの指針の検討

4.1 原田の知見の検討

前章で挙げた方針は大学の授業を前提に設計したものであり、小学生を対象とする場合、同じでは済まないことは当然である。方針を小学生向けに改訂することを考えるとして、「何が、なぜ同じでは済まないか」に関する知見が必要である。

本稿ではそのような知見として、Viscuit の開発者である原田が、(おもに児童・生徒を対象とした) プログラミング学習に関しての見解を書いているブログの中から、筆者が目にしたうちでとくに重要な指摘と考えるものを取り上げ、その内容に基づいて検討する。[*5]

19-5-28: なぜ繰り返しや条件分岐はプログラミング入門に向かないか [1] — この記事では、プログラミングで繰り返しや条件分岐を教える場合、変数に違う値が入っているから毎回違う動作が起こるのであり、プログラミング言語における重要な概念である変数を教えることが必須であると指摘している。また、タートルグラフィクス等では変数を教える代わりにタートルの状態で済ませることから、当面はよくても頭打ちとなる、とも指摘している。

筆者もこの意見に賛成であり、L3 を前提とする以上、小学生であっても変数とその書き換えは必ず理解すべき事項として取り入れるべきだと考える。その場合、それを始めるのは「初回」であり、最初に離陸する時に精選した事項の 1 つとして扱い、以後常に使い続けるのがよいと考える。変数の使用を前提とするなら、制御構造 (繰り返し、分岐) は普通に取り入れられる。

18-11-21: プログラミングの教育と大衆化の違い [2] — この記事では、プログラミングの教育と大衆化を区別すべきということを論じている。教育の目的は「仕事や研究で使う言語で自由にプログラミングが作れる」こと、大衆化の目的は「それまで専門家しかできなかったプログラミングが専門知識がなくても自由に作れる」こととしている。そして、小学校の義務教育では大衆化をすすめるべきと述べている。原田のいう教育の目的が筆者のいう「離陸」と同一であるのは興味深い。

具体例として、お化け (多くの類似した例題での「ロボット」「タートル」に相当) を特定の経路に沿って進ませるという課題に対し、(1) 上下左右の移動命令を並べる、(2) 前進、右回転、左回転の命令を並べる、(3) 進ませたい経路に沿って矢印を配置し矢印の方向に進ませるプログラムと合わせる、の 3 案を示し、(1)(2) は従来型のプログラミングに近いがより難しく教育に相当し、(3) はやって欲しいことを直接示せるという点で優れていて大衆化に相当する、と述べている。

筆者も、Viscuit が変数という抽象化なしに「多くの具体例をルールとして列挙することで複雑なプログラムが (手間はあっても) 作れる」ことと、それによって変数のあるプログラムに「挫折した (ここは推測)」人たちが楽しく作りたいものをプログラムできていることには敬意を持っている。

もともと前項のような変数の概念を扱うには、一定の抽象化能力とそれを可能とする発達段階が不可欠であり、原田のいう「教育」はできるとしても小学校後半となるように思える。そして中学校では汎用言語によるコーディングが求められることから、小学校後半で「全員ではないとしても一部は」変数による離陸を目指してよいのでは、と思う。

18-10-06: 理想的なプログラミング言語 [3] — この記事では、プログラミング教育が「できないからできるへの変化」、より具体的には「抽象化の習得」を目指していて、それに対し Viscuit は「やりたいと思ったことをやってみてできる」ことをめざすという点で「子供にとってより理想的である」と述べている。

ここまで折り合いをつけてきたが、ここに至っ

[*5] 定期購読しているというわけではないので、見逃した重要な指摘があればご容赦いただきたい。

て筆者と原田の立場は決定的に対立している — 原田は「抽象化は難しいから目指さないのがよい」、筆者は「抽象化を身に付けて汎用言語に進んで欲しい」という立場なので。そこでもう少しだけ筆者の立場を言わせてもらえば「小学校では算数の計算といい国語の漢字といい山のようにトレーニングをしているので、そこを少し削って (発達段階的に大丈夫なところで) 抽象化の習得に振り向けた方が全体として幸せなのでは」ということである。もっとも筆者がカリキュラムを決めるわけではないので、本稿も「やるとしたらこうすればできるのでは」という提案が趣旨である。

4.2 小学校段階における指針の適用

本節では前節の検討も参考にしながら、前章の方針を小学校向けに改訂するとしたらどのようにするかを検討していく。

P1: 離陸ファースト — 前節で述べたように、変数の概念をきちんとマスターさせることを初期の目標とする。このため、初回を「手続き定義+変数を用いた計算」のようにすることが考えられる。手続きを扱うのは、抽象化を目的とする以上、手続きは避けて通れず、それなら最初から扱うのがよいと考えるためである。

P2: 多様な水準の演習問題 — 生徒ごとに違う問題をやると教える側の負担が高まる可能性があるが、だいたいの生徒は易しい問題から順にやるので、はじめの方の問題ができていない生徒はできている生徒に教えてもらうように「教え合い」を活用することで対応できると考える。

P3: 演習の重視 — もともと、大学から前に遡るほど体験・実習的な要素が高まるので、小学生を対象としたプログラミングでは「演習の重視」は取り入れやすい。ただしそのためには、各自が使える実習環境 (PC) が必須となる。

P4: 苦手な人の学びを促す — この方針については、原則は大学の場合と同じとしても、小学校ではそれほど先行している生徒がいるとも思われないので、P2 にある「教え合い」程度の運用で対応できるのではと思われる。

P5: 「プログラムが書ける」目標の明示 — この方針についても、原則は同じであるが、小学生ではそもそも「知識を記憶して試験で得点する」ような擦り込みはまだなく、普通にプログラムが書けることを目標として提示すれば済むものと考えている。

P6: 書き方の自由度の明示 — これについては、大学の場合と同様、例題において複数の記述方法があることを示すことで対応できると考える。

P7: プログラムは自分の頭で作り出す — 方針としては同じとして、手段として絵や動画を課題としたいが、汎用言語で絵や動画まで到達するにはその前段階の内容が多くなるという課題がある。かといって、単純な数値の操作は課題としてあまり魅力的でない。そのため、この方針については維持するとしても、その実現方法が分かっていない状態だと考える。

全体を俯瞰すると、P1〜P2 と P6 については小学生向けに題材を工夫して設計することで対応でき、P3〜P5 については、むしろ小学校段階では大学生に見られるような問題が少なく、特に頑張らなくても自然に対応されると期待する。

それに対し、P7 は未解決の課題が残されている。自分から作り出したいと考えるような「目標設定」の部分は L0〜L2 のさまざまな環境で工夫されている部分であり、そのような手助けのない状態で適切な目標設定が行え、効果的な学びにつなげられるかどうかが、L3(汎用言語での実現) においてキーポイントとなる可能性がある。

5. 小学生対象テキストの試作

一般論だけでは具体的な細部の検討ができないので、前節で述べた指針の改訂版をもとに、小学生 (5〜6 年生想定) 対象のプログラミング入門コースのテキストを試作してみた。テキストそのものは付録に掲載している。

使用言語は L3 を前提に JavaScript を採用しているが、小学生対象の場合細かい操作でのつまずきは避けたいため、ブラウザ上で動作する JavaScript 実行環境である「JavaScript 練習ページ」(図 3) を前提とした [*6]。そして P7 のために絵を作り出し

[*6] このページは所属大学のリテラシ科目でも簡単に高水

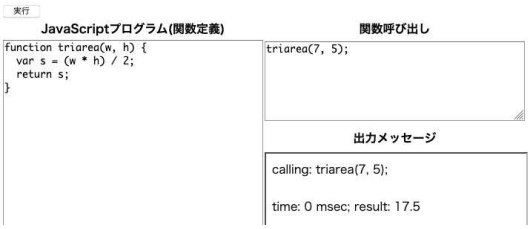


図 3　ブラウザ上の JavaScript 実行環境の例

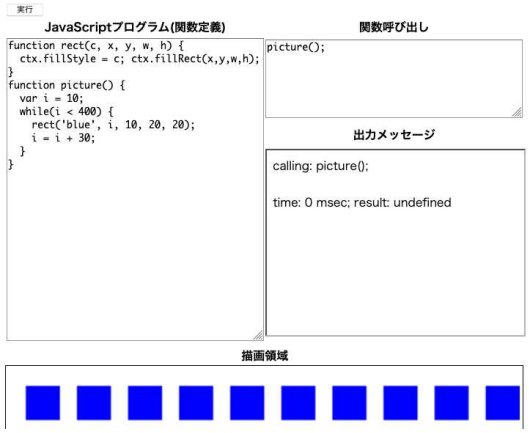


図 4　canvas を併設した実行環境

たいことから、canvas 機能に簡単にアクセスできる機能を追加した「canvas つき練習ページ」(図 4) も使用している。

テキストはカリキュラムの骨格を示しアイデアの検討に用いることを目的としているので、最小限の内容とし、言語やプログラミング全般に関する詳細の説明は省いている。その上で、canvas つき練習ページで絵が作れるところまでを 3 回ぶんの内容として制作した。

3 回ぶんの内容を表 1 に示す。P1(離陸ファースト) のため、取り扱う題材を最小限とし、その題材については十分な練習を行うように構成している。たとえばループは while のみに限定し、描画も fillRect のみを扱っている。P2(多様な水準) のため、演習問題には難しいものも含めている [*7]。P3(演習重視) については、テキストの説明を最小限とし、すぐ演習に進めるように作成した。P4(苦手な人対応)、P5(プログラムが書ける) については、前述のように小学生対象であることと、短いコースであり評価を含めないことから、特別の配慮はしていない。P6(自由度) については、多くの例題で複数の書き方を最初から提示している。P7(自分で作り出す) については前述のように解かれていない課題であるが、ここでは最小限のステップで絵の描画まで進み、最後の課題を自由制作とするように構成している。

表 1　試作テキストの構成

回	内容
#1	変数による値の計算
	例題: 三角形の面積の計算
	題材: 関数・パラメタ・変数・代入・式
	演習: 様々な計算を行う関数を書く
#2	制御構造 (if,while)
	例題: 2 数の差 (の絶対値)
	題材: if 文、条件、場合分け、再代入
	演習: 2(3) 数の最大、正負零の枝分かれ
	例題: カウントアップ
	題材: while 文、反復、変数の加算
	演習: 様々な規則による計数ループ
#3	canvas グラフィクス
	例題: 正方形を並べる
	題材: canvas、矩形の fill、下請け関数
	演習: 矩形の様々な配置
	例題': RGB 色指定 (前の例の拡張)
	題材: 色指定、連続的な色変化
	演習: 自由制作

実際にテキストを制作してみて苦労した点は、小学生だとまだ算数の段階で、数値の計算の題材で使えないものが多い点である。[*8]

また、最短距離で絵の出力まで到達するため、取り上げる題材を精選した。とくにループで while 文のみとし for を扱わないことや、本来はループの次に扱う配列を省略したことが特徴であるが、汎用言語を学ぶとすればこれらは当然扱う必要がある (絵の題材の先で扱うことでももちろん構わない)。

準言語を体験してもらう目的で使用している。

[*7] 演習 2 の x^8、演習 3 の 3 数の最大、演習 4 のフィボナッチ数列、演習 5 の市松模様 (2 重ループ) などが高度な問題に相当する。

[*8] たとえば大学では枝分かれの最初の例題は絶対値にしているが、絶対値は小学校算数の範囲外である。

6. 議論とまとめ

本稿では筆者がこれまで大学初年次プログラミング科目で取り入れて来た指針を整理し、それらが小学校向けにも有効であるかを検討した。とくに重視している指針「離陸ファースト」について、離陸にも複数のレベルがあることを指摘し、L0〜L4 の 5 レベルを提案した。指針についてはとくに L3(汎用言語による離陸) をおもなターゲットにしていると位置付けられる。

その後、P1〜P7 の 7 つの指針について具体的に説明し、小学校向けの改訂の方向について検討した。結果としては、多くの指針についてはそのまま適用可能であり、一部については小学校レベルでは特別な配慮は不要そうである、と考える。一方、P7(自分でプログラムを作り出す) について、L2(教育用言語) で提供しているような「すぐ動かして見られる」環境については汎用言語だとそこに到達するまでの手数が長くなるという問題があると考える。

これらの検討に基づき、小学校 5・6 年生を対象とした JavaScript による入門コースのテキストを作成し検討した。このコースでは絵を作り出すという段階まで 3 回程度で到達するが、そこまで「すぐ誰でも」進めるとは言えないかも知れない。ただし、中学校ではテキスト型言語を用いることとなっているので、このような形の教材によるテキスト型言語入門コースをどこかの段階で受けることは有用である可能性がある。

謝辞 Viscuit の開発者である原田康則さんには、本テーマに関連する内容について、たびたび有益なご意見をいただいて来た。また本稿では原田さんのブログの内容を検討材料として多く取り入れさせて頂いた。ここに感謝します。

参考文献

[1] 原田康則, なぜ繰り返しや条件分岐はプログラミング入門に向かないか, ビスケット開発室 2019.5.28. https://devroom.viscuit.com/2019/05/28/post-1870/

[2] 原田康則, プログラミングの教育と大衆化の違い, ビスケット開発室 2018.11.21. https://devroom.viscuit.com/2018/11/21/post-1659/

[3] 原田康則, 理想的なプログラミング言語, ビスケット開発室 2018.10.6. https://devroom.viscuit.com/2018/10/06/post-1615/

[4] 情報入試研究会, 試作問題#001 問題 (略解つき), 第 2 問, 2013. http://jnsg.jp/wp-content/uploads/2013/01/joshin2012aki-johoexam-shisaku.pdf

[5] 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.

[6] 久野 靖, 何のためにプログラミングを学ぶの? そしてどのように?, 夏のプログラミングシンポジウム 2016: 教育・学習, Sep 2019

[7] 久野 靖, プログラミング入門科目の指針と実践例 (前編), 情報処理, vol. 60, no. 3, pp. 244-247, Feb 2019.

[8] 久野 靖, プログラミング入門科目の指針と実践例 (後編), 情報処理, vol. 60, no. 6, pp. 541-545, May 2019.

付　　録

A.1 コース例: JavaScript によるプログラミング入門

A.1.1 基本方針

具体的な教材・コース内容の例として「JavaScript によるプログラミング入門」を示す。想定学年は小学校 5・6 年であり、次のことを基本方針としている。

- 変数と手続きによる抽象化を自分のツールとして獲得することが目標。制御構造は変数と組み合わせることで導入。
- 自分の思ったものを作る題材として canvas グラフィクスを採用。
- つまづきを少なくするため Web ブラウザ上で動く実習サイトを使用。

本来であれば (大学生向けなら) 制御構造の次に「配列」を取り扱うが、ここではスキップし、#1: 変数と手続き、#2: 制御構造、#3: グラフィクスの 3 回ぶんの内容として示す。[*9]

[*9] 実習には「JavaScript 練習ページ」「同 canvas 機能つき」を用いる。同ページは当面の間以下に置く。 https://www.edu.cc.uec.ac.jp/~ka002689/sprosym19/

A.1.2 #1: 変数による値の計算

JavaScript ではプログラムは「関数」と呼ばれるかたまりを組み合わせて作っていきます。関数とは、ひとまとまりの機能を作り出すプログラムに名前をつけたものと考えてください。たとえば、三角形の面積を計算する、という例で考えます。そのための関数は次の形になります。

```
function triarea(w, h) {
  var s = (w * h) / 2;
  return s;
}
```

1 行目の「`function triare(w, h)`」で関数の定義を始め、`triarea` という名前をつけています (triangle の area—面積—のつもりでつけました)。`w` と `h` は関数に与えるパラメタ (計算のつど様々に変化させられる値) です。ここでは三角形の面積なので、`w` は底辺の長さ、`h` は三角形の高さのつもりです。続く「`{ ... }`」が関数の範囲で、その中には「文 (命令)」がいくつも入れられます。文は並んでいる順番に動作していきます。

最初の文は「`var s ...`」で、まず「変数」を用意します。変数とは、計算に使う値を覚えておく「いれもの」であり、パラメタも変数の一種として働きます。変数に値を入れる (覚えさせる) ことを「代入」と呼びます。「`s = 計算式;`」により、値を計算して、その結果を変数 `s` に代入しています。[*10] このプログラムの場合は計算式は `(w * h)` つまり `w` と `h` を掛けて (`*`はかけ算を表す)、結果を 2 で割るという内容です。なお、代入は何回でもおこなえ、最後に入れた値が覚えられています。また、変数の名前を書くことで、そこに覚えられている値が取り出されます。[*11]

2 番目の文は、`s` に入っている (前の文で計算した) 面積を「結果として返す」働きをします。関数で計算した結果を見るには、return で返させる方法と、(後で出てきますが)「`document.writeln(...)`」で途中結果を表示させる方法とが使えます。

ところで、return の後に書くものは任意の計算式でよいので、上の例では変数 `s` の値を取り出して返していましたが、次のプログラムのように面積を計算してそのまま返す (変数 `s` を使わない) ように書くこともできます。

```
function triarea2(w, h) {
  return (w * h) / 2;
}
```

このように、「正しい」プログラムはいく通りもの方法で作ることができるのが普通です。

プログラムを動かすには「練習ページ」を使ってください。練習ページでは、「関数定義」のところに作成した関数を書き込み、「関数呼び出し」のところに「`triarea(7.0, 5.0)`」のようにパラメタとして与える値をつけた関数 (関数呼び出し) を書き込みます。そこで「実行」ボタンを押すと、「出力メッセージ」のところに計算結果が表示されます。

演習 1 練習ページに上で示した「三角形の面積計算」関数の好きな方を打ち込み、動かしてみなさい。プログラムの一部をわざと違うように直した場合どうなるかも観察しなさい。

演習 2 次のような関数を作りなさい。呼び出して実行して結果を確認すること。

a. 2 つの数を受け取り、和を計算する。(よかったら差、積、商も)。
b. 演算子「`%`」は剰余 (割った余り) の演算である。剰余を計算する関数を作って結果を試しなさい。
c. 円柱の底面の半径と高さを受け取り、体積を計算する。円周率は `3.14` であるものとしてよい。
d. 数値 x を受け取り、x の 8 乗 (8 回掛けたもの) を計算する。かけ算の回数を少なくできるとよい。
e. 何か自分が面白いと思う計算を行う。

A.1.3 #2: 制御構造

前回のプログラムでは、関数のそれぞれの文は上から順番に 1 回ずつ実行されていました。しかしプログラムでは、「条件に応じて、ある文を実行したりしなかったりする」「条件が成り立っている間、ある文を繰り返し実行する」などの処理も必要となります。これらの処理のことを (実行の流れを制御することから)「制御構造」と呼びます。

1 つ目の制御構造として「枝分かれ」を記述する文である if 文を学びます。

```
if(条件) {
  条件が成り立つ場合の処理
} else {
  上記以外の場合の処理
}
```

[*10] JavaScript では「`=`」は「代入する」という動作を表し「等しい」という条件は「`==`」のように等号 2 つで表します。

[*11] JavaScript では加減乗除算はそれぞれ`+`、`-`、`*`、`/`で表し、式を 1 行で書くために、適宜 `(...)` で囲みます。

条件としては、a > b (a が b より大きい) などの条件を書くことができます。[*12] また、「上記以外の場合」の処理が何もないときは、次のように else とそれに伴う処理を書かないでよいです (1 行に書いても先の例のように複数の行に分けても構いません)。

```
if(条件) { 条件が成り立つ場合の処理 }
```

if 文の例として、2 つの数値 a, b がどれだけ離れているか (たとえばものさしの上で 2 つの数値を指定したとき、その間がどれだけ空いているか) を計算する関数 dist を作ってみました。

```
function dist(a, b) {
  var d;
  if(a > b) {
    d = a - b;
  } else {
    d = b - a;
  }
  return d;
}
```

ここで d が「どれだけ離れているか」の度合いです。まず変数 d は用意しておき、if 文で枝分かれします。もし a と b で a が大きいなら、a-b を計算して d に入れます。それ以外なら、b が大きいかまたは a と b が等しいので、b-a を計算して d に入れます。結果としては d の値を返せばよいです。

ところで、おなじ問題は次の書き方にしてもできます。

```
function dist2(a, b) {
  if(a > b) {
    return a - b;
  } else {
    return b - a;
  }
}
```

これは、いちいち変数 d に入れる代わりに、計算した結果を直接 return で返します。もう 1 つ、次のものはどうでしょうか。

```
function dist3(a, b) {
  var d = a - b;
  if(b > a) { d = b - a; }
  return d;
}
```

これは、とりあえず a の方が大きいとして、d に a-b を入れます。それから if 文に入り、もし b の方が大きかったら、そのままでは困るので、b-a を計算して d に「入れ直し」ます。変数には値を何回でも入れられることに注意。ところで、a と b が等しければ? そのときは最初の計算で 0 が入るでしょうから、そのままでよいわけです。

同じことをする正しいプログラムが何通りにも書けることがお分かりいただけるかと思います。

演習 3　上の例題の好きなものを練習ページで動かし、さまざまな値で動作を確認しなさい。納得したら、次の関数を作りなさい。

a. 2 つの異なる数 a, b を受け取り、より大きい方を返す。
b. 3 つの異なる数 a, b, c を受け取り、最大のものを返す。
c. 数 x が正なら「'positive'」、負なら「'negative'」、零なら「'zero'」という文字列を返す。[*13]
d. 3 つの異なる数 a, b, c を受け取り、中央の (最大でも最小でもない) ものを返す。

もう 1 つの制御構造として、「繰り返し」を記述する文である while 文を学びます。

```
while(条件) {
  繰り返し実行する処理
}
```

while 文では「処理」が何回も実行されるので、少しのプログラムで沢山の計算を行うことができます。その実行のされ方は次のようにイメージするとよいでしょう。

- 「条件」を調べる (成立)。
- 「処理」を実行。
- 「条件」を調べる (成立)。
- 「処理」を実行。
- …
- 「条件」を調べる (不成立)。
- 繰り返しを終わる。

「終わる」条件を指定したくなりやすいのですが、「終わらない (繰り返し続ける)」条件を指定する、というところがポイントです。

それでは例題として「整数 n を与えたら、1, 2, ... n

[*12] 正確には、>(より大)、>=(以上)、<(より小)、<=(以下)、==(等しい)、!=(等しくない) の 6 種類の比較が使えます。

[*13] 文字列は文字の並びのことで、両側を'...' または "..."で囲むことで表せます。

とカウントする」というのを見てみます。

```
function countup(n) {
  var i = 1;
  while(i <= n) {
    document.writeln(i);
    i = i + 1;
  }
}
```

`document.writeln(...);` というのは、プログラム実行の途中で文字や数値を出力します (練習ページだとメッセージ領域に現れます)。最初 `i` を `1` にしたので、`1` が表示されます。次に、`i+1` を計算するので、`2` が計算されて、それを i に代入するので (「=」は代入動作を意味することに注意)、`i` には `2` が覚えられます。n としてある程度大きな値を指定したら、2 はそれ以下でしょうから、再び `document.writeln(...);` に進み、`2` が表示されます。次の `i+1` は `3` が計算され、それを `i` に覚え直します。このように次々と数値が表示されていきますが、たとえば `n` として `10` を指定したら、`10` を表示したあと、`i` が `11` になったらそれは「n 以下」ではないので、これで繰り返しが終わります。このプログラムは `document.writeln(...);` で次々に出力するのが仕事なので、return で値を返すことはしていません。

演習 4　カウントの例題を動かし確認しなさい。納得したら、次の関数を作りなさい。

a. 1 からでなく 0 からカウントし、n の 1 つ手前でやめる。
b. 1, 3, 5, ... と奇数だけカウントし、n は表示しない。
c. 1, 2, 4, 8, ... と倍々の値で n 未満のものを表示する。
d. n, n-1, ..., 1, 0 と 1 つずつ減らしながら 0 までカウントする。
e. 1, 1, 2, 3, 5, ... のように最初の 2 つは 1、以後は「直前の 2 数の和」を n 未満の範囲で表示する。

A.1.4　#3: canvas とグラフィクス

ここまではプログラムの出力は (計算結果を表す) 文字ばかりでしたが、もっと興味を持てる題材として絵の出力 (グラフィクス) を取り上げます。JavaScript を Web ページ上で動かす場合、canvas と呼ばれる四角い出力領域を使ってグラフィクスを出力することが一般的です。このため、練習ページにも canvas がすぐ使えるように「canvas つき練習ページ」を用意しました。このページにはあらかじめ canvas が用意されていて、変数 `ctx` からその機能が呼び出せます。まず最低限として次の 2 つを使います。

- `ctx.fillStyle = '色指定';` — 塗りつぶしの色を指定。
- `ctx.fillRect(`x`, `y`, `w`, `h`);` — 長方形を指定色で塗る。

色の指定はとりあえず「`'red'`」「`'blue'`」など色の名前を文字列として指定すると思ってください。長方形の指定ですが、x と y は canvas の左側と上側から長方形までの距離、w と h は長方形の幅と高さです。[*14]

それでは、複数の正方形を並べて描くという例題を見ましょう。

```
function rect(c, x, y, w, h) {
  ctx.fillStyle=c; ctx.fillRect(x,y,w,h);
}
function picture() {
  var i = 10;
  while(i < 400) {
    rect('blue', i, 10, 20, 20);
    i = i + 30;
  }
}
```

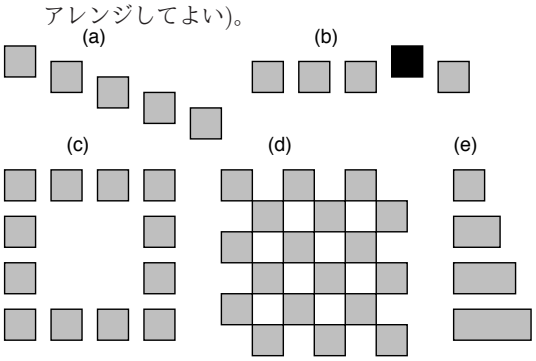
まず `rect` という関数がありますが、これは色 c を使って x, y, w, h の位置/大きさの長方形を塗りつぶします。内容は上で説明した canvas の機能を順に呼び出すだけですが、すぐ次の絵を作る関数を分かりやすく書くために、この関数を作りました。これが打ち込めたところで、「`rect('red', 50, 50, 80, 40);`」などとして呼び出して動作を確認してみてください (色、位置、大きさを色々変えてみてください)。

次の関数 `picture` が絵を作り出す本体で、変数 i を 10, 40, 70, 100, ... と増やしながら、横から i、上から 10 の位置に幅・高さともの 20 の青い正方形を描いています。このプログラムを動かした様子は図 4 のようになります。このように、1 つの四角を描くだけなら手で描いた方が速くても、多数の四角を正確に規則的に描くような場合はプログラムを作る方が便利なのです。

演習 5　「多数の正方形」の例題を動かしてみなさい。動いたら、色や正方形の位置、大きさをさまざま

[*14] 大きさの単位は「ピクセル」で、おおよそ 20 ピクセルで 1 センチくらいだと思えばよいでしょう。

に変えて試してみなさい。納得したら、次の図のような絵を作ってみなさい (個数や配置は好きにアレンジしてよい)。



ところで、色が「真っ赤」「真っ青」などではあまりきれいではありません。もっと様々な色を出すには、ちょうど絵の具を混ぜ合わせるように、赤/緑/青の色をさまざまに混ぜ合わせます。そのような指定のため、関数 rgb を作りました。

```
function rgb(r, g, b) {
  r = r%256; g = g%256; b = b%256;
  return 'rgb('+r+','+g+','+b+')';
}
```

この関数は赤/緑/青の強さを 0〜255 の範囲の数値で指定しますが、間違って大きな数値を渡してしまった場合でも大丈夫なように 256 で剰余を取ってから使います (つまり 255 を越えると 0 に戻ります)。そのあと、「'rgb(*r*,*g*,*b*)' という文字列を作って返しますが、この文字列は canvas の色指定としてそのまま使うことができます。[*15] この rgb を使って、先の例題の四角を描くところを次のように直してみました。その結果できた絵は図 A·1 のようになりました。

```
rect(rgb(i, 100+i, 100), i, 10, 20, 20);
```

図 **A·1**　色を変化させてみる

演習 6　色が連続的に変化するような絵を作ってみなさい。

演習 7　自分が楽しいと思う絵を描くプログラムを作りなさい。

[*15] 「+」演算は数値に対しては足し算をしますが、文字列に対しては「文字列をくっつける」操作になります。

質疑

Q.　レポートで解説はどのように要求しているか。(中西)

A.　説明せよとだけ要求。説明する過程があればよいと考えている。

Q.　小学生で英語で書かせるのはよいのか。function とか。(〃)

A.　小学校でも英語やりますよね。関数の説明は最初からする。おまじないはできるだけ避ける。

Q.　for は教えず while だけにするのはよい。(〃)

A.　変数と書き換えという教えたい内容が直接見えるのでよいと思う。

Q.　抽象と具体の対立だが、抽象化は中学では必要だと考えている。小学校でやるにしても全員でなく一部でよい。(原田)

A.　変数というものは全員に学んでもらってよくないかという提案。

Q.　全員にやらせる最低線の中に一部のできる子向けがあると嫌いになる子ができるのでは。(〃)

A.　小学生の経験があまりないのでそれは分からない。変数と手続きに厳選するのでどうかという提案。

Q.　小学生はともかく、先生が好きになってくれないとだめ。

A.　それは難しい問題ですね。

Q.　コーディングをやるという提案だが、概念を学ぶのにコーディングをやってもよいという提案か。(山中)

A.　コーディングは目的ではないが、概念を学ぶという目的の手段としてコードを書くというのは別によいと考えている。

Q.　むかし小学生に LOGO を教えた経験があるが、変数よりはタートルなどのものの方がよかったという経験。(山口)

A.　タートルは定評があるが、ここでは重要な概念で汎用性もある変数を教えたいという提案。

Q.　中学校に入ると数学で変数を学ぶがその前だと難しいかも。(〃)

A.　計算した値を入れておく、という方が具体的で単純。プログラミングの方を先に学び後から数学の変数というより抽象的なものに進むのでもよいかも。

Q.　数学の先生からの反発があるかも。(〃)

A.　そういう社会的な問題は難しいですね。