# プログラミング体験に適した処理系とその特徴

久野　靖[1]

2015.8.3

[1]筑波大学ビジネスサイエンス系

# はじめに

本講座は「SSR: 大学の講義を聞こう 2015」と「情報処理学会教員免許更新講習: プログラミング体験に適した処理系とその特徴」を兼ねる形で実施します。

演習については、東京大学のご好意により、教育用計算環境を使用して実施します。今回取り上げているプログラミング環境である「ビスケット」「ドリトル」ともに多くの環境で動作しますが、それぞれ一定の制約はありますので、持ち返って授業に使用される場合は動作するかどうかチェックしてください。

進め方ですが、第 1 章 (概論) についてはレクチャーとディスカッションを行い、第 2 章 (ビスケット)、第 3 章 (ドリトル) については授業案を説明した後、実際に授業内容に相当する演習を行っていただきます。ただし、本当の授業の時間を使ってしまうと 1 日で収まらないので適宜短縮し、なおかつ進み方にもよりますがそれぞれ最後の方は演習は割愛するかも知れません。

全体として、単に講座を聞くということでなく、演習により体験していただき、それに基づき気づいたことをディスカッションして頂く、という形で進めます。最後に免許更新講習の方は試験となりますが、SSR 側の方はその時間も討論をおこないます。では、1 日という限られた時間ですが、よろしくお願い致します。

# 第1章　プログラミング学習の基本的枠組

この章では、プログラミング学習の目的について確認した上で、その目的を達成するためにはどのような形の学習が望まれるのか、どのような形は避けるべきなのかについて、検討してゆきます。併せて、望ましい形の学習に適したプログラミング環境の必要性や要件についても取り上げます。

## 1.1　プログラミング学習の目的

皆様は、学校教育においてプログラミングを取り上げることの理由ないし「目的」をどのように考えていらっしゃるでしょうか。たとえば、わが国では IT 人材が何万人不足している、みたいな話はよく出て来るので、次のように考えられるかも知れません。

× より多くの IT 人材を育成するため。

しかしそれは「主要な目的としては」正しくないと考えます。たとえば、保護者の方たちがご自分の子供達を IT 人材 (情報技術者) にさせたいと考えているでしょうか? そういうことは、皆無とは言いませんがあまりないと思われます。そのような、社会的合意のできていないことを学校教育の目的にすることはできませんね。[1]

実際、プログラミング教育反対論の多くは「子供達が情報技術者になるわけでもないのに…」という形のものですから、情報教育の目標を上記のものと「誤解した上で」それに反対していると言えます。

なお、それ以外の主要な反対論は「もっと子供達に学んでもらう重要なことが色々あるだろう、それなのにプログラミングを追加するなんて」です。それに対する私の反論は「プログラミングを追加していない現状において、それらの色々な『重要なこと』の学習は必ずしもうまく行っていないですよね? そしてプログラミング学習はその状況を変える手段となる可能性もあります」というものです。これについては少し後で触れます。

さて、それではプログラミング学習の目的は何であるとするのがよいのでしょう? それは 1 つには次のものだと考えます。

○ コンピュータや情報技術について原理から理解するため。

プログラムを書いてみなくてもコンピュータの構造や仕組みの講義を受ければ原理が理解できる、と思いますか? 表面的な原理であればそうかも知れませんが、「結局どういうことができてどういうことは大変なのか」というのを身を持って分かるためには、プログラムを書いてみる以上によい方法はない、というのが筆者の考えです。

とくに、プログラムで記述するためには、手順を完璧に詳しく、曖昧さなく定める必要があるわけですが、このことを納得するのにはプログラムを書いてみた経験がほぼ不可欠と言っていいでしょう。

実は、わが国のソフトウェア開発は他国と比べて生産性が低いのですが、それはソフトウェアを発注する顧客側の企業に上記のことを身をもって知っている人がいなくて、「何を作って欲しいか」を開発側にうまく説明できてないことが大きな要因だと筆者は考えています。

[1] もちろん、プログラミングを学んだ結果、この方面に関心を持つようになった児童・生徒が情報技術者になりたいと思うのは自然なことですので「副次的な」目標としては別に否定しません。

上のことがらとも関係しますが、プログラミング学習のもう1つの目的としては、次のものが挙げられるでしょう。

○ 手順的な問題の記述や問題解決方法を理解し使えるようになる。

プログラミングを全員が学んだとして、実際にどれくらいまで自分でプログラムを書けるようになるかはそれぞれの人次第ということはあります。しかし、問題やその解法の手順をステップで記述していく、という考え方は非常に広い範囲に応用でき、実際に役に立ちます。

さらに、ある程度のところまで自分で自分が欲しいと思うプログラムを作って問題を処理できることは、それ自体でとても価値があります。これまでのわが国では、プログラムで何かを解決しなければならない時は、専門家に頼んで作ってもらう、というのが普通でした。しかしこれには時間もお金も掛かりますし、上に述べたようなことで、うまく問題が伝えられず、思うようなものができてこないことも多いのです。

世界各国がプログラミング学習に力を入れているのは、かなりこの面が大きいと考えます。ある問題に遭遇したとき、自力でプログラムを書いて処理してしまえる人と、専門家に頼んで何週間も待たないと先に進めない人では、明らかに勝負になりませんから。

そして、今日の小学生が大人になったとき、彼らの就く職業の65%は今日まだ存在していない職業だ、という説もあります。その「新しい」職業には情報技術が必須であり、プログラミングを書けないとやって行けない、というケースもかなりあると思われます。

ここまでに挙げた「○」の理由2つはある意味、かなり「実利的」なものだと言えます。しかし実は、もっと重要な次の理由がある、というのが筆者の考えです。

◎ プログラミングを学ぶことは楽しく、そのために、より深く学んだり考えたりする力が養われる。

現在、学校教育の内容は「試験で正解するために知識や解法パターンを覚える」方向に偏っていると言われます。それも、小学校から中学、大学入試が控えている高校と進むにつれてひどくなるとも。PISAテストの結果などでも、日本の生徒は成績は悪くないものの、考えたり書いたりする部分が相対的に劣っていると指摘されています。

これに対しプログラミングは、同じ動作をするコードでも複数通り書けるため、「唯一の正解」からは遠い世界にあります。また、学習に際して「自分が作ろうと思うものを作る」という形を取ることで、それぞれが自分の目標に向かって試行錯誤しながら学ぶようにできます。

人間は「自分の目標に向かって試行錯誤」するときに最もよく学ぶので、これは大切なことです。つまり、プログラミングの学習はごく自然に「能動的学習 (アクティブラーニング)」の形態を取ることができ、今求められているような授業改革の一環となることができると考えるわけです。

## 1.2 プログラムの本質

さて、ここまで散々プログラミングについて語ってきましたが、「プログラムとは何か」についてまだ説明していませんでした。プログラミングとはプログラムを作る行為ですから、これでは片手落ちでしょう。

実は「プログラムとは何か」についても、色々な捉え方があって簡単には決められないところがあるのですが、ここでは次の要件を満たすようなものがプログラムである、ということにします。

(a) 決まった規則によって文字を並べたり画面上に要素を配置したものであって、

(b) その並びや配置に応じて (並びや配置のカタチに対応して) 定まったやりかたで、

(c) コンピュータが一連の動作を自動実行する。

このうち、最初の2つ (a)、(b) については、我々が普段つかう言語 (自然言語) でも同じことが言えますから、それを例にして見てみましょう。

This is a pen.

という英文を見ると、a〜z の決まった形のアルファベットの文字が使われ (英字でない文字が混ざっていたら英語として読めない)、それが並んで this とか is とか決まった意味を持つ単語が構成されていて (並びが違ったら別の単語になってしまう)、それらがこの順に並んでいることにより、「これはペンです」という決まった意味を表しています。また、語順を変えて

Is this a pen?

とすれば「これはペンですか?」という疑問文になります。

このように人間が使う言語がこのような「定まった形ごとに決まった意味に対応している」ことから、それがコンピュータに対する命令つまりプログラムでも使われるようになった、と考えていいでしょう。

では (c) の「自動実行」についてはどうでしょうか? たとえば「電卓」は計算ができますが、コンピュータとはだいぶ違いますよね。電卓で計算するときは、何をどう計算するか、人間がボタンで逐一、指示して行きます。ですから計算の順序は人間が自由に判断して調整できるのですが、代わりに人間がボタンを押す速さでしか計算できません。

コンピュータは非常に高速に処理が進められますが、それを活かすためには「計算の手順」をあらかじめ (a)、(b) に基づいて用意しておき、それを参照しながら (c) 自動的に (つまり人間の介在なしに) 実行するしかないのです。これがプログラムの本質、というふうに考えます。

このように、プログラムはコンピュータに能力を発揮させるためにやむをえず今日のような形になっているものではありますが、それを実際に扱うことでたまたま、冒頭で述べたようなよいことが得られる、というふうに考えられるわけです。

## 1.3 プログラミング学習の目指す枠組み

さて、冒頭で述べたような「好ましい学習」がプログラミングにおいて成立するためには、注意すべきことがいくつがあります。たとえば、次のようなことは避けるべきです。

× プログラミング言語の機能や文法を逐一説明し、覚えることを要求する。

× テキストに載っている例題のみを「正解」とし、そこから少しも外れないことを目標とする。

× 生徒が興味を持ちにくいような課題を天下りで与えて書かせる。

「そのまま暗記」や「想定正解との一致」を求めてしまうと、プログラムが持つ柔軟性と、そこに由来する「学習者に自分で考えることを促す」側面が失われてしまいます。そして「興味を持てないような課題」は、「学習者が自発的に学ぶことで深く学べる」面を損なってしまいます。

そうなる代わりに、最初の手がかり (書き方や考え方など) は教師が導くとしても、「人によってさまざまな書き方があり唯一の正解はない」ことを繰り返し伝え、「生徒にとって興味深い題材」を取り上げ、「自分で選んだり考案した課題に向かって努力する」ように導くことで、プログラミングの学習題材としての特徴を引き出すことができます。

そのために、まず最初の目標とすべきことは、「自分でこうしたいと思ったらそのようにプログラムが作れる」ことだと考えます。これを私たちは「離陸する」と呼んでいます。離陸さえできたら、そこからは学習者がそれぞれ思ったように飛んで行くことで、その先の学びに自然につながるからです。まとめると、筆者が考える「望ましい枠組み」は次のようになります。

○ ある範囲内で「こうしたいと思ったこと」を「プログラムとして作れるようになる」(離陸)

○ 自分で「こうしたいと思うこと」を決め、それを達成しようとする (目標のある遊び)

○ 教員の仕事は「学習者が達成する上で必要な素材を必要に応じて提供する (遊びの手助け)

この枠組みでは、教員は決して「自分の教えたいことを教えようと」してはいけないことになります。それは学習者が自発的に求めていることと違うかも知れませんから。そうではなく、学習者が「こうしたい」と思ったときに必要なことを、求められた時に、提供してください。人は「自分の目標に向かって試行錯誤」するときに最もよく学ぶのですから。

**open questions**

- 皆様の近辺では、プログラミング学習の目的は何であるというふうに理解されていることが多いでしょうか?
- 「プログラムを書いてみなければコンピュータや情報技術のことは本当には分からない」という主張は納得できますか? 納得できないとすれば、どんな異論があるでしょうか?
- 「ソフトウェアを発注する人が情報技術のことを分かっていないために、大きな無駄が生じている」という主張は納得できますか? 納得できないとすれば、どんな異論があるでしょうか?
- 「誰もがある程度まで、自分の必要とするものを自分でプログラムできるようになるべき」という主張は納得できますか? 納得できないとすれば、どんな異論があるでしょうか?
- 「プログラミングを学ぶことは楽しく、そのために、より深く学んだり考えたりする力が養われる」という主張は納得できますか? 納得できないとすれば、どんな異論があるでしょうか?
- アクティブラーニングの利点について納得していますか? また、アクティブラーニングの題材としてプログラミングは適しているという主張についてはどうでしょうか?
- 「決まった規則に従い要素を配置すると、その配置に応じて定まったやり方で、コンピュータが動作を自動実行」というプログラムの定義は納得できますか。また、高速なコンピュータを役立てるには「自動実行」が必要という説明は納得できますか。
- 「そのまま暗記や正解との一致を求めてはいけない」という主張は納得できますか? 納得できないとすれば、どんな異論があるでしょうか?
- 「こうしたいと思ったようにプログラムが書けるようになる (離陸)」という概念についてはどう思いますか。難しすぎる目標だとか思いますか?
- 「教師は教えたいと思うことを教えようとしてはいけない」「学習者に求められたときにはじめて提供」をどう考えますか。カリキュラムに従って授業内容を進めて行くという考えと相反するように思えますか?

# 第2章　ビスケットによるプログラミング学習

ビスケット (Viscuit) は原田康徳さん (デジタルポケット) が前職の NTT 研究所時代に開発したプログラミングシステムであり、文字を使わないという特徴から、小学校低学年からの使用に適しています (「ビスケット塾」などでは入学前児童のコースを開催していました)。本章ではビスケットを用いた学習カリキュラムを実際に体験しながら、1 章で取り上げたプログラミング学習の目的との対応についても検討して行きます。

## 2.1　基本動作: 絵を動かす

### 2.1.1　概要と授業案

ビスケットは「自分で描いた絵を画面で動かす」ことが基本となっています。動作対象の絵を組み込みのツールを使って自分で描くというのは、比較的以前からある Squeak eToys と呼ばれる教育向けシステムに起源があると思われますが、次のような得失があります。

- ○ 自分が作成した絵が動かせることはモチベーションを持たせる効果がある。
- ○ 自分が考えたプログラムに「必要な絵」は探してくるより自分なりに描いた方がぴったりする
- × 絵の作り込みに熱中してプログラミング学習という目的まで到達しないことがある。
- × 意図した絵がうまく描けないことでモチベーションが下がることがある。

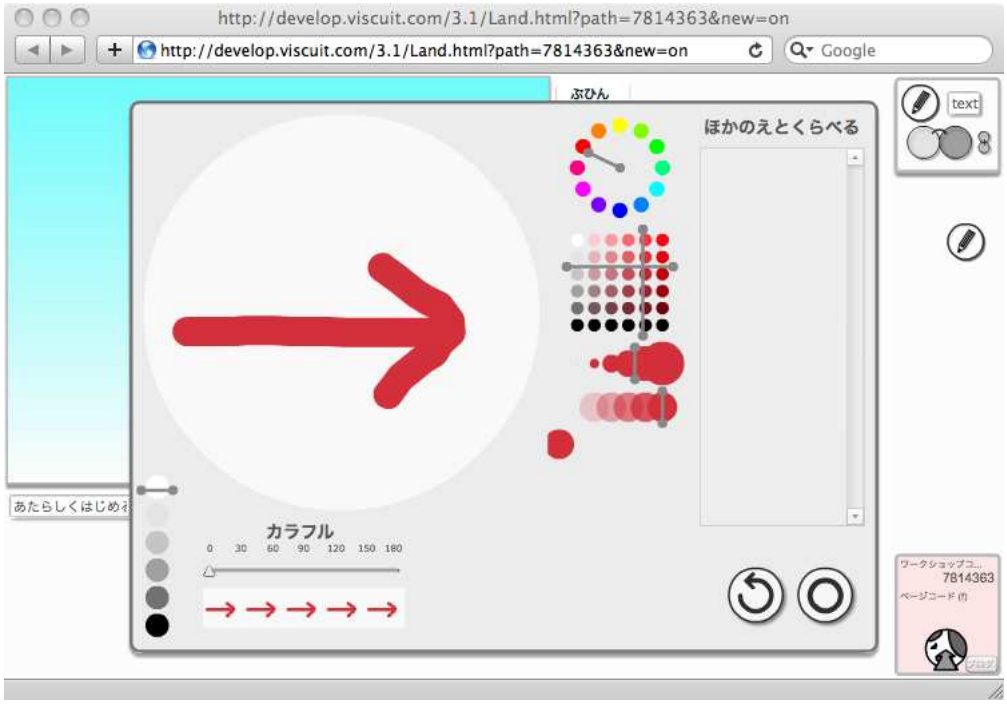


図 2.1: ビスケットの描画ツール

ビスケットでは、お絵描きに手間を掛けてしまったり上手下手が気になったりしにくいように、シンプルな絵を太いペン先で描き、その際に美しい色合いが選べるようなツールを工夫しています。ここから先は指導案の形で見て行きましょう (表 2.1)。

表 2.1: ビスケット 1 時間目: 絵を動かす

| | |
|---|---|
| 導入 (5 分) | ビスケットは絵を動かすプログラムを作るシステムであることを説明。プログラムを体験してもらい、コンピュータの特徴を知ることが目的。 |
| 絵を描く (10 分) | ビスケットのサイトを開く (http://www.viscuit.com/)。「ビスケットの基本を練習しよう」の「やってみる」を選ぶ。右上の箱からえんぴつアイコンを選択すると絵のツールが出る (図 2.1)。色やペンの太さが選べる。戻すアイコンで取り消し。最初は矢印とかマルとか四角とか簡単なものにしてもらう。完成したらまるアイコンで終了。自分のプログラムに使いたい絵 (部品) をいくつでも描いてよい。 |
| 絵の配置とめがね (10 分) | 絵を描き終わって「○」を押すと「ぶひん」の所にその絵が現れている。そこからステージ (色のついたところ) に繰り返しドラグすることで、部品を何個でも置くことができる。右上の箱から「めがね」をプログラム置き場 (白いところ) にドラグする。いくつもドラグでき、不要なら箱に戻せる。「×」がついているのは「動作していない」めがね。めがねの左と右の円に部品の絵を 1 個ずつドラグしてくると動作する。右の部品を中心位置より少し右にずらすと、ステージ上の部品も右に動く。「このめがねは、ステージにあるその部品を右に動かすというプログラムになっている。」めがねの調整のしかたで動きは上下左右さまざまにできる。回転させたければステージ右下にあるスイッチ (ボタン) の「まっすぐ」を「曲がり」で切り替えてから、めがねの右側の部品の端をつかんで回転させる。 |
| 各自の構想作品制作 (20 分) | 部品もめがねもいくつでも作って使えることを考え、自分で「どのような部品をどのように動かす」か計画して紙にスケッチし、それに応じたプログラムを作ってみよう。やってみて気がついたことを書き留めておこう。 |
| まとめ (5 分) | 気がついたことや感想を自由に言ってもらう。ビスケットのめがねは部品を決まったやり方で動かすことと、このようなものを「プログラム」と呼ぶことは確認する。 |

なお、指導案中には書いていませんが、いちど作った作品は保存して後でさらに直すことができます。そのためには、画面右下の保存ボタンのところにある「ワークショップコード」「ページコード」をメモします。そして、作業中に時々保存ボタンを押して保存します。保存したものを取り出して動かしたり直したい時はビスケットのページ http://www.viscuit.com/の下の方にある「ワークショップの入口」を選択し、「ワークショップコード」「ページコード」を打ち込めば最後に保存したところに戻れます。

ここで取り上げている授業案では、1 時限で「絵が好きなように動かせる」ことまでを扱っています。先に述べたように「好きな絵を描く」「その絵を好きなように動かせる」ということがポイントで、具体的にどんな絵、どんな動き、というところは自分で決めるわけです。

### 2.1.2 何を気付いてもらうか

どのような授業でもそうですが、プログラミング体験をおこなう場合はとくに、「体験したこと」の中に「どのような重要なことが含まれているか」をうまく気付いてもらうことが大切です。いちばん良いのは子どもたちの中から気付きが生まれることですが、視点を変えないと気付けないことやうまく言語化しにくいことも含まれるので、教員がヒントを出したりすることも考えられます (が、覚えさせて試験に出すとかは本末転倒なのでやめましょう)。

この節で扱った授業案の内容を体験したときに遭遇し得る重要な事項としては、次のものが考えられます。

- 1 つの絵を何個でもキャンバスに並べられる — コンピュータ上の情報はコピーは極めて容易なので、1 つ作ったら沢山出して使うのはほとんど手間なしにできます。いっぽう「ちょっ

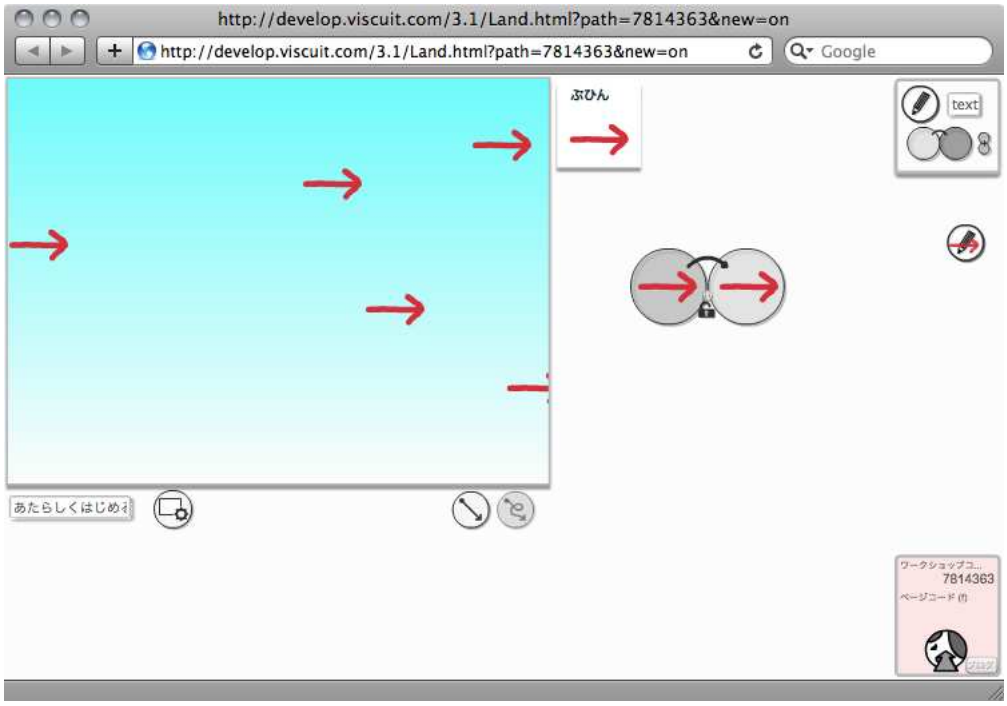


図 2.2: ビスケットのめがねで絵を動かす

とだけ違ったもの」にするには人間がその違いを作らないといけないので、それなりの手間が掛かります。

- 1 つの「めがね」でキャンバス上の多数の絵を動かすことができる — コンピュータは非常に高速であり、1 つのプログラムで多数のデータを扱うことは容易である。人間が絵を動かすのであれば、絵が 2 個、3 個になったら 2 倍、3 倍の手間が掛かり、その分だけ大変ですが、コンピュータでは「どれだけ増えてもほとんど同じ」です (コンピュータの働きによりものすごく増えた時は遅くなることも)。
- 1 つの「めがね」が動かす絵はどれもまったく同じ動きである— コンピュータのプログラムは厳密に定まった動作を寸分の違いなく行うので、同じプログラムによって操作される絵はまったく同じように動かされるし、プログラムを変更したら一斉にその動きが変化する。人間が多数の絵を動かすとしたら、同じように動かそうとしても少しずつ「ぶれ」がありますが、コンピュータではそのようなことはありません。
- 「めがね」が×印 (エラーがあり動作しない状態) になると、そのめがねが動かしていた絵は一斉に停止する — 1 つのプログラムが多数のデータを操作するということは、そのプログラムに間違いや問題があったら、その間違いや問題は多数のデータに一斉に影響します。普通のものや機械では、使っているうちに一部が故障するとしても一斉ではないわけですが、プログラムの場合は「故障が発現するときは一斉に発現」するので大きな影響をもたらすことがあるわけです。

こうして見ると、ビスケットで簡単な絵を動かしてみるだけでも、コンピュータやプログラムの性質についてかなり学ぶことがあることが分かります。

## 2.2 めがねによる書き換えと条件判断

### 2.2.1 概要と授業案

前節の部分では、部品 (絵) は沢山あったとしても、それぞれバラバラに動いているだけでした。しかし実は、めがねの左と右に別の絵を入れることで「ある絵を別の絵に取り替える」ことがで

きます。たとえば、大きい心臓の絵と小さい心臓の絵を用意し、めがねを 2 つ用意して「大きい心臓は小さい心臓に」「小さい心臓は大きい心臓に」取り替えるようにすると、ドキドキする心臓が作れます。そして、絵が入れ替わる時の配置によって動いて行くような効果も作れます。

さらに、左右のめがねに入る絵は 1 つずつでなくてもいいのです。ビスケットでは設定によって絵の増減を許すこともできますが、初歩のモードでは絵が増減しない設定になっているので、左右に入る絵の個数は同じにします (これが守られていないと×になり動作しません)。その状態でも、沢山の面白いプログラムが作れます。これらのテーマを扱う指導案を見て見ましょう (表 2.2)。基本となる例題は「尺取り虫 (図 2.4) です。尺取り虫が縮んだ絵と伸びた絵を交互に取り換えることで、這って進んで行く様子を表すわけです。

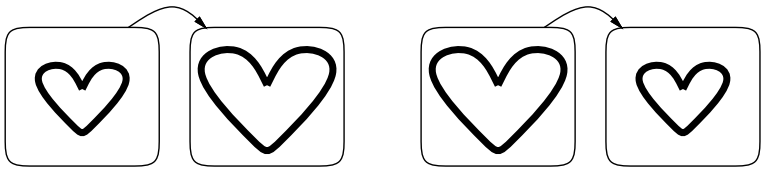

図 2.3: ドキドキする心臓

表 2.2: ビスケット 2 時間目: 絵の書き換え

| | |
|---|---|
| 導入 (5 分) | ビスケットのめがねには左右に違う絵を入れることができることを説明。ドキドキする心臓の例を黒板などで説明。 |
| 切り替わる絵の作品 (15 分) | ドキドキする心臓を作ってみてもらう。うまく行ったらもっと他の「切り替わる絵」の作品を計画して、制作してもらう。途中の適当な時点で「尺取り虫」の例題をめがねを隠して見せ、こういうのも作れるよと紹介し作れるかなと言う。 |
| 複数の絵の書き換え (5 分) | 「尺取り虫がボールを蹴る」のめがね (図 2.5) を見せ、このようにめがねの左右に複数の絵 (ただし絵の数は同じ) が入ることもできると説明。このプログラムはどんなことが起きると思うかあてさせ、実際に動かして見せる。自分で試すときはボールや尺取り虫を増やし、尺取り虫をいろいろな角度にすることを促す。 |
| 各自の構想作品制作 (20 分) | 複数の絵の書き換えが可能なことを用いて、自分で「どのような動きをする作品」か計画して紙にスケッチし、それに応じたプログラムを作ってみよう。やってみて気がついたことを書き留めておこう。 |
| まとめ (5 分) | 気がついたことや感想を自由に言ってもらう。めがねに複数の部品を入れ、複数のめがねを使うことで複雑な動きも作れることは確認する。 |

授業案の中に出て来る「尺取り虫」のプログラムを図 2.4、さらにこれに追加して「ボールを蹴る」機能を入れるプログラムを図 2.5 に示す。尺取り虫が「伸びる」時は後足の位置が揃うように、逆に「縮む」時は前足がの位置が揃うようにすることに注意。

### 2.2.2　何を気付いてもらうか

ここの部分でもプログラムが持つ特徴について、複数の重要なことが含まれています。具体的に揚げておきます。

- 異なる絵に次々に書き換えて行くことで「連続した一連の変化」を実現できる — プログラムが行うことはごく単純な「1 回の書き換え」であってもそれが「複数ある」ことで連鎖して長い動作や無限に続く動作が作り出せます。この「単純な動作」を「多数連鎖して実行」がコンピュータの本質なわけです。

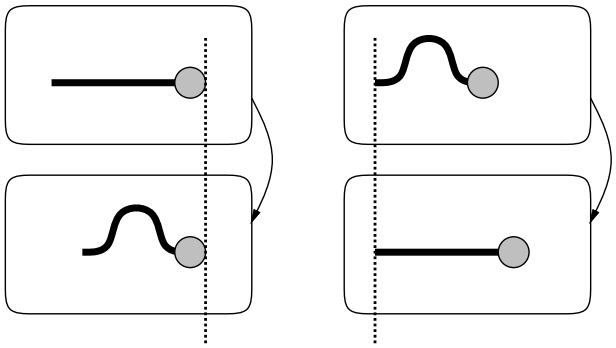

図 2.4: 尺取り虫のプログラム

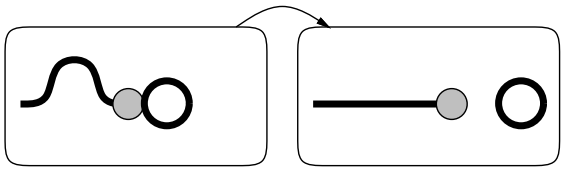

図 2.5: 尺取り虫がボールを蹴る

- 尺取り虫が動いて見えるためには、位置の揃え方がポイント — 「心臓がドキドキ」のように絵の中央を合わせていると、縮んだ絵と伸びた絵が交互に切り替わるように見えるだけで、進んで行くようになりません。つまり、プログラムの「構造 (論理)」は同じでも、意図した動作 (この場合は効果や見え方) を達成するためには「位置の調整」すなわち「適切なデータ」が組み合わせられてなければいけないわけです。これは一般のプログラムでも多く見られることです (プログラムの構造だけまねしてデータを精査しないため意図したことができないというのは、プログラミングの初心者によく見られる現象です)。
- 縮んだ尺取り虫で「ボールがある場合」「ない場合」の 2 通りに枝分かれすることで条件判断ができる — ここで行っていることは「ボールがあったら蹴る」ですが、この「あったら」というのは条件判断であり、条件に応じてさまざまな処理を組み合わせることで複雑なプログラムが構成できます。
- まったく同じ絵 (の組み合わせ) がめがねの左にある場合に思い通りの動作にならないかも — 人間はその場その場で判断して選択するので曖昧さをあまり気にしませんが、コンピュータは厳密な動作をおこなうので、曖昧さが苦手です。曖昧さがある場合「動作を停止する」「任意に 1 つ選んでいつもそれだけ使う」「毎回さいころを振って決める」など、いく通りかの方式が考えられますが、それがプログラムを作った人の期待と一致するかどうかは分からないところです。
- めがねを 1 つ増やしただけでこれまでよりずっと複雑な動作になる — 「ボールを蹴る尺取り虫」を動かすと、1 匹が複数のボールを蹴ったり (ジャグラー?)、複数で 1 つのボールを取り合ったり (サッカー?)、複雑な動きが生まれることが分かります。たとえば、これまで 4 通りの動作をしていたものに、枝分かれを 1 つだけ追加することで、最大で 8 通りの動作に分かれることになります (実際は 4 通りのうちのどれか 1 つとか 2 つにだけ関係する枝分かれを追加することもありますが)。つまり、プログラムが少し複雑になるだけで、プログラムの動作は「倍々で」複雑さを増すことができるわけです。私達が普段さまざまな仕事に使っているプログラムはめがねが 1 万とか 2 万とかあると思えばいいでしょう。

## 2.3　インタラクション

### 2.3.1　概要と授業案

ここまでにやってきたプログラムでは、作った絵が箱庭的に動き観察 (鑑賞) できるだけでした。これに対し、インタラクション (対話) のあるプログラムでは、ユーザ (プログラムを使う人) がプログラムの実行中に入力操作を行い、それによってプログラムの動作に影響を与えることができます。ゲームのプログラムなどがその典型例です。

ここから先では、これまでに無かった機能を使うため、ビスケットの起動時に「ぜんぶいり」を選択してください。マル 1、マル 2、矢印などのアイコンが現れていますが、めがねにこれらのアイコンと同じマークをつけておくと、そのメガネはこれらのアイコンをクリックした時だけ動作します。これによって入力が実現されるわけです。また、歯車アイコンを選択するとメニューが出て、ビスケットの動き方を調整できます。そのほかにも、ここでは説明しない多数の機能がついていますが、それらについてはビスケットの説明などを直接見てください。

では、インタラクションを扱う授業案を見てみます (表 2.3)。基本となる例題は「1 ボタンを押すと発射するロケット (図 2.6) です。点火していない状態で 1 ボタンをクリックすると書き換えが起こり、点火したロケットに変化します。そして点火したロケットは上方に移動するので飛んで行きます。

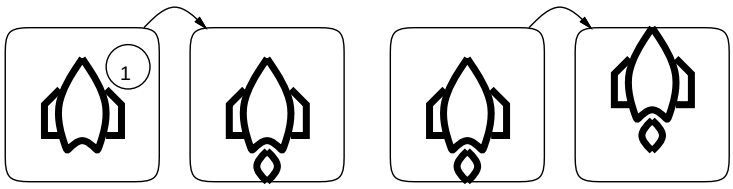


図 2.6: 発射されると飛ぶロケット

表 2.3: ビスケット 3 時間目: インタラクション

| | |
|---|---|
| 導入 (5 分) | ぜんぶいりの画面を見せ、マル数字や矢印の機能を説明する。1 ボタンを押すと発射されるロケットの例を説明。 |
| インタラクションの体験 (10 分) | ロケットでもそれ以外のものでも、ボタンを押すと動き出す作品を作ってもらう。また、左右矢印で左右の位置を変化させるなども試してもらう。 |
| ゲームのデザインについて説明 (5 分) | 画面上方に星の絵を横に動かし、ロケットが当たったらバクハツの絵に変更するとどうかな、ロケットが障害物に当たると故障とかどうかな、とヒントを与える。2 つの絵 (ロケットと星) を 1 つの絵 (バクハツ) にするには絵の数が変わるので、歯車アイコンで設定を出し、「ぶひんの数が変化」をチェックすることを説明。また、部品が増えてよくしたら、砲台 (図 2.7) なども作れることを説明。そのほか、ロケットや玉が上に出て行ったら下から出てこないように「たてにつながる」をオフにすることも説明。 |
| ゲームの製作 (25 分) | 簡単なゲームを紙にスケッチしてデザインさせ、それから実際に制作させる。完成したら 2 人で互いに相手のゲームを体験させる。 |
| まとめ (5 分) | 気がついたことや感想を自由に言ってもらう。 |

### 2.3.2　何を気づいてもらうか

ここまでの授業案で気づいてもらいたいことがらとしては次のものが挙げられます。

- インタラクションによる動的な変化 — これまでの動く絵だと動き方はあらかじめすべて決まっていたので、タイミングによるランダム性はあっても一定の流れで実行が進んでいまし

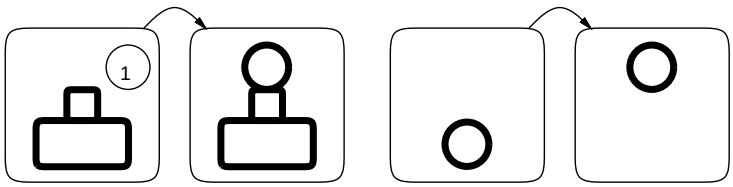


図 2.7: 砲台から玉が発射され飛んでいく

た。これに対し、ユーザの入力によるインタラクションがあると、1 つのプログラムからこれまでよりずっと多様な動きが生み出されます。

- 「使いやすさ」の出現 — インタラクションがあるということは、ユーザは何らかの意図を持ってそのインタラクションを起こすので、プログラムの動作がユーザの意図に沿っているか否か、という次元がプログラムに生まれます。すなわち、「使いやすい」「使いにくい」の違いが生まれることになります。そしてその違いは、プログラムの構造的な設計によることもありますが、タイミングのような「データの調整」によることも多いです。
- ゲームの易しさ/難しさ — ゲーム全体として見た場合、単に「敵を倒しやすい」のがいいゲームではありません。易しすぎるゲームはつまらないし、逆に難しすぎるゲームも目的が達成できずにつまらないです。つまりゲームバランスという「ちょうどよいところ」を探す行為が必要になります。難しさはしばしば、部品の動く速さに関係しているので、ビスケットではめがねの調整によりバランスが調整できます。
- 自分以外の人の感じ方 — 自分にとって「わかりやすい」「つかいやすい」「バランスのいい」ゲームプログラムだと思えても、他の人にやってもらうと、必ずしも同様に感じてもらえるとは限りません。これらの評価基準はかなり個人差があり主観的なものでもあるので、作者と別の人で違うことは当然です。他人に使ってもらうプログラムの場合は、作者にとってよいかではなく、(当然) 使ってくれる人にとってよいかで評価や調整をおこなう必要があります。

## 2.4 シミュレーション

### 2.4.1 概要と授業案

シミュレーションとは実際に実験するのが大変だったり難しかったりすることを模型などで模擬的に試してみることを言います。コンピュータはここまでに見てきた通り、多数の「もの」を同じ規則にしたがって動作させることが得意なので、シミュレーションに適しています。

なお、多数の「もの」を動かしてみるシミュレーションは離散イベントシミュレーション (discrete event simulation) と言いますが、そのほかに数式で組み立てたモデルに従って計算するようなシミュレーション (数値シミュレーション) も多く使われます。そちらは数式があつかえる中学・高校レベル以上の話題となるでしょう。今回は「もの」を動かす方を扱います。

さて、簡単な例として「ジャンケン」を考えてみます。「グー」「チョキ」「パー」の部品が出会ったらジャンケンをして、図 2.8 のように、負けた方は消え、勝った方が増えます (ということは、部品の数は変わらないわけです)。このほかに、動かないと出会えないので「グー」「チョキ」「パー」それぞれが違った方向に動くめがねを用意します。

これで、キャンバス上に次のような状況を作って動作させたら、どうなるでしょうか。

- 多数の「グー」「チョキ」「パー」が配置されていて、そのうち 1 つの数が多い状態。
- 多数の「グー」「チョキ」「パー」が配置されていて、その数が同じである状態

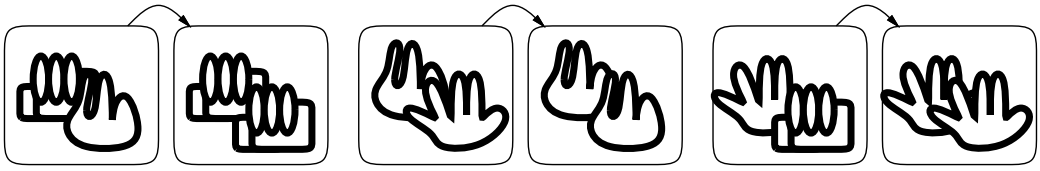

図 2.8: じゃんけんの規則をプログラムする

実際にこれらの状態でシミュレーションを実行してみると、その結果は多くの人にとって予想外のものであると思われます。つまりシミュレーションは、ただ考えて予想しただけでは分からないような、新たな知見をもたらしてくれるわけです。

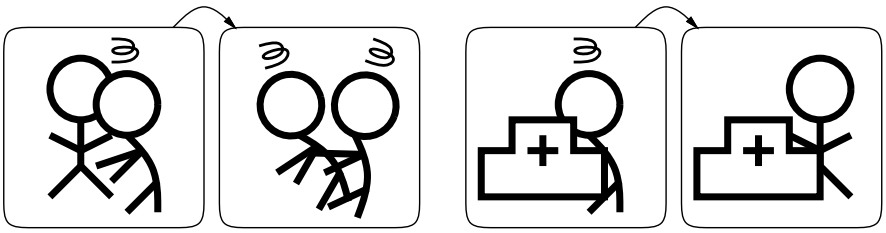

図 2.9: 病気の感染のプログラム

シミュレーションのもう 1 つの例として「病気の感染」を取り上げます。これは図 2.9 左のように、健康な人と病気の人が接触すると、健康な人も病気になる、というものです (この他に健康な人が横方向、病気の人が縦方向に動くめがねを用意して、両者が接触しやすくします。また、感染した後は 2 人が別の方に動いて離れて行くようにめがねの右側の絵は回転させます)。これで、多数の健康な人の中に一人病気の人を入れるとどうなるでしょうか。この結果は比較的予想しやすいと思われます。

次に、図 2.9 右のように病院を作り、病気の人が病院と接触すると健康になるようにします。病院がどれくらいあったら、病気の蔓延を防げるできるでしょうか。たとえば、シミュレーション開始時に病気の人と病院の数が同じだったら大丈夫でしょうか。

ここまでの内容を授業案としたものを表 2.4 に示します。

表 2.4: ビスケット 4 時間目: シミュレーション

| | |
|---|---|
| 導入 (5 分) | シミュレーションという言葉を知っているか問いかける。どんな意味か、なぜコンピュータと関係あるのか簡単に説明。 |
| じゃんけんの説明 (5 分) | 沢山の人が「グー」「チョキ」「パー」になり、出会ったら負けた側は勝った側になるというゲームを説明し、めがねで作って見せる (動くだけのめがねも必要)。動作はさせないでおく。「1 つだけ多い場合」「3 つとも同じ数の場合」でどうなると思うか言ってもらう。 |
| じゃんけんのシミュレーション (10 分) | 実際に個人やグループでじゃんけんのシミュレーションをしてもらい、先の問いかけの答えをまとめてもらう。 |
| 病気の感染の説明 (10 分) | 病気の感染のシミュレーションを、今度は「これは健康な人」「これは病気の人」とだけ言い、あとはめがねを作ってみせる。何が起こると思うか予想させる。病院の治療も同様にする。「どれくらい病院があればいいと思うか、病人と同じくらいでいいか」問いかける。 |
| 病気の感染のシミュレーション (15 分) | 実際に個人やグループで感染のシミュレーションをしてもらい、先の問いかけの答えをまとめてもらう。 |
| まとめ (5 分) | 気がついたことや感想を自由に言ってもらう。 |

### 2.4.2 何を気づいてもらうか

ここまでの授業案で気づいてもらいたいことがらとしては次のものが挙げられます。

- シミュレーションの価値 — 規則だけを見ても結果が予想できないようなことは色々あります。そのようなことに対して、結果を予測する手段としてシミュレーションはとても有効で価値があります (世の中の現実の問題の多くにあてはまります)。
- 指数的増加の強力さ — 病気の感染は「1 人が 2 人、2 人が 4 人…」のように「倍々」で増えて行きます。これを「指数的増加」といいます (「$2^n$」のように表せるから)。指数的増加は目に見える所ではあまり起こらないので分かりにくいですが、コンピュータ上では簡単に起こるのでそのことを体験しておくことは価値があります。
- 線形増加と指数的増加の対比 — 線形増加とは一定時間ごとに一定量ずつ増える増え方です。今回の場合は病院が $n$ 個あるとその病院の数に比例して病気の治療が起きるので健康な人の増加は線形増加に近い形になります。指数的増加は (病気の人が一定以上に増えた時) 極めて急速に増えるので、線形増加では (病院による治療では) 間に合わなくなるわけです。このような、線形増加と指数的増加の対比はコンピュータによる処理でよく現れます。
- プログラムを読めることの価値 — 今回の授業案では、プログラムを動かす前にプログラムだけを見せてどう動作するか考えさせています。プログラムを「読める」ことは、プログラムの論理を考えたり、それをもとに新たな動作を作り出して行く基礎となります。他人のプログラムを読んで理解し改良することで、学習のサイクルをスムーズに進めていけるようになります。

## 2.5 二進法

ビスケットの最後の話題として、二進法を取り上げます。これは小学校では実習には難しいかも知れないので、授業案はなく、内容の説明のみにしてあります。

私たちが普段使っている十進法では、数字として「0」〜「9」までを使い、足し算は図 2.10 左の表によっています。ずいぶん沢山の項目がある表ですが、これを私たちは小学生のときに暗記させられるわけです。1 桁どうしの足し賛の結果、上の桁への繰り上がりが「1」ある場合があります。

| + | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 4 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 5 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 6 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 7 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 8 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 9 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |

| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 10 |

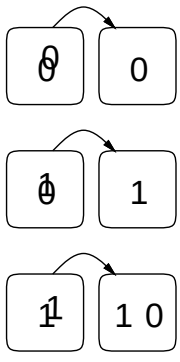


図 2.10: 十進法と二進法の足し算規則

一方、コンピュータが使っている二進法では、数字として「0」「1」だけを使いますから、計算の規則が非常に簡単です (図 2.10 中)。規則が簡単だと、電子回路として構成しやすく、また高速化しやすいので、それでコンピュータでは二進法を使うわけです。図 2.10 右では、この足し算の規則をビスケットのめがねで表したようすを示しています。

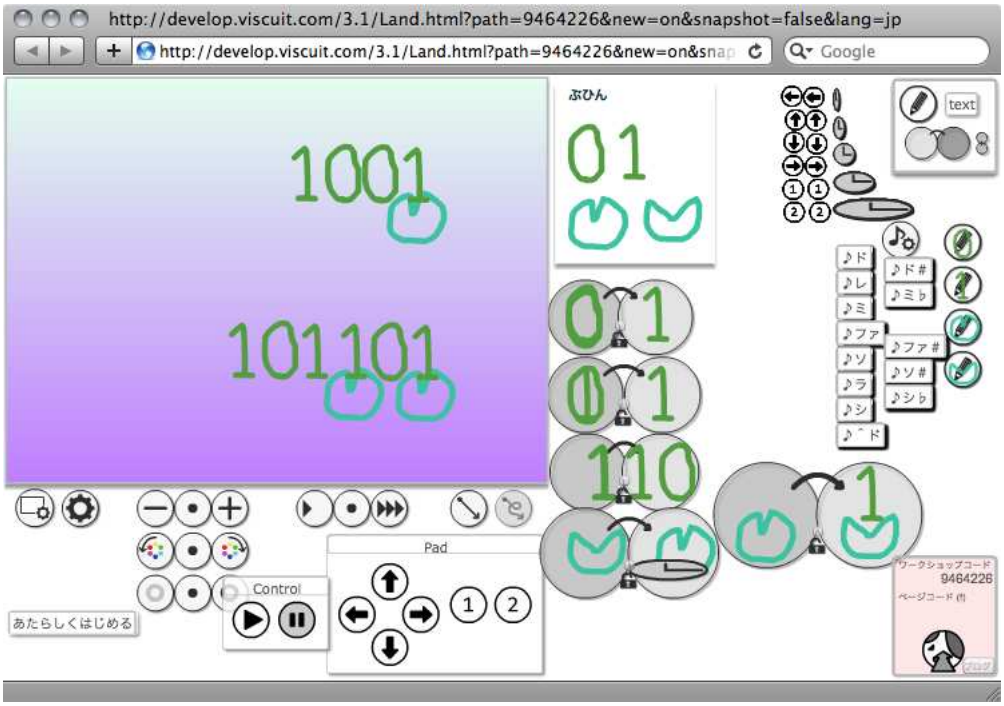


図 2.11: ビスケットで作った二進法のカウンタ

これを実際にビスケットで動かしたようすを図 2.11 に示しました。このような絵の配置が崩れてほしく無いものの場合には、歯車アイコンを出して「なめらかな動き」のチェックを外してください。

このプログラムの最初の 3 つのめがねは、二進法の足し算の規則そのままです (0 と 0、1 と 1 が重ねてあるところは見づらいですが)。あとの 2 つのめがねは、パクパクする丸で、口が開いたときに真上に「1」を生成します。時間を少しゆっくりにする必要があるので、片方のめがねには時間待ち (時計のアイコン) が入れてあります。

これを動かすと、キャンバスの上のように、1 ずつ値が増えて行くカウンタが作れます。数値は当然、二進法で表現されています。また、下のように「1 の桁と 4 の桁」に 1 が生み出されると、5 ずつ値が増えて行くカウンタになります。

このように簡単にできるのは、二進法の規則の数が少ないからだということに注意してください。もしこれを十進法でやるとしたら、先程の表全部に相当するめがねが必要となり、かなり大変であることがお分かりいただけるかと思います。

## 2.6　ビスケットのまとめ

このほか、ビスケットでは図 2.12 のように音を出す機能なども使うことができ、多様な作品を作ることができます (これも画面の配置が崩れてほしくないので「なめらかな動き」のチェックを外しています)。

一方で、本資料で扱ったように、その動作の中にはコンピュータが特徴として持つ様々なことがらが含まれていることが分かります。これらをうまく組み合わせて「作品作り」と「気付き・学び」をともに達成できることが、ビスケットの題材としての興味深く有用なところであると考えます。

ビスケットの特徴はここまでに見てきたように多数ありますが、「文字を使わない (読むことすら必要としない)」というのもその 1 つです。このため、まだひらがなを学習中の小学校 1 年生や入学前児童でも扱うことができます。

そしてもう 1 つの特徴として「遊べる (=離陸する) までの敷居が低い」ことも挙げられます。自分で自分なりの絵を描いて、めがね 1 つでそれが動かせるわけですから。自分の作品を自分で考

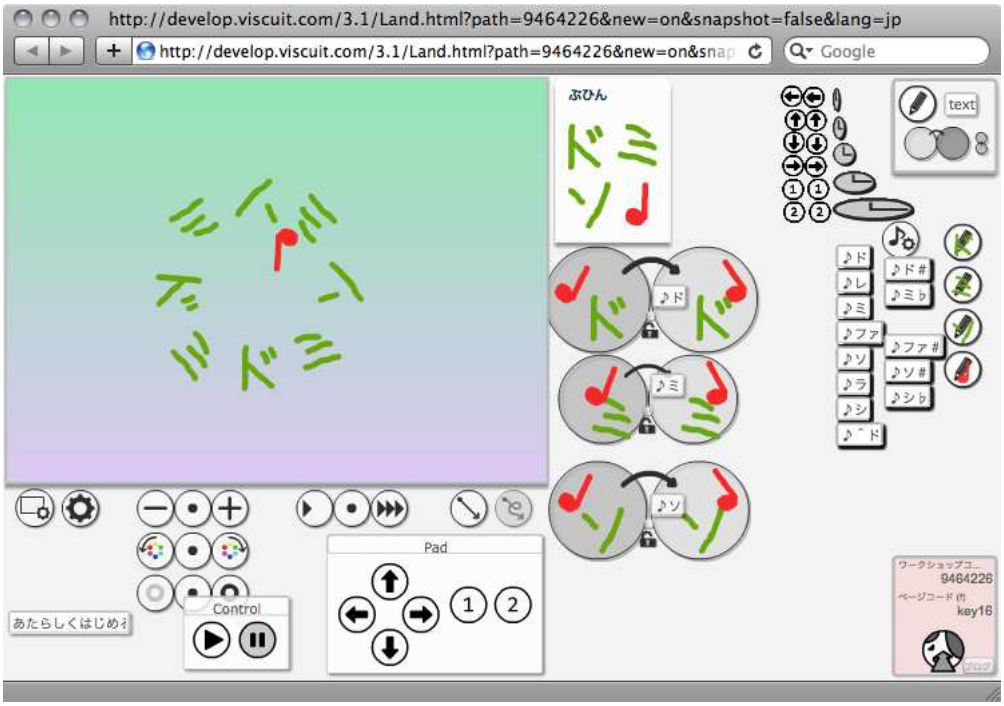


図 2.12: ビスケットで作ったオルゴール (?)

えて作ったり直したりする、という「遊び」が「そのために懸命に考える」活動を自然とうながすわけです。

個人的には、このこと (遊び＋考える) が、プログラミング学習において一番重視して欲しいところです。もちろん、ここまでに挙げてきたさまざまな「気づき」も興味を持ってもらえたらよいことなのですが、それを「教えたり覚えたり」することが目的になってしまわないように注意したいと思います。

### 謝辞

(合) デジタルポケットの原田康徳氏には、本資料で取り上げている題材についてご教示を頂き、その資料ならびに授業案での利用をご快諾頂きました。ここに感謝します。

### open questions

- ビスケットによるプログラミング体験は普通 (?) の言語の体験とだいぶ違うと思いますか? 同じだと思いますか? 違うとしたらどちらがいいの?
- ビスケットにかこつけてコンピュータの性質をずいぶん挙げましたが、納得が行きましたか? ちょっと違うと思いましたか? コンピュータについて知るためのビスケット、という目標設定はどれくらい良いでしょう?
- ビスケットを用いて「状態遷移」「順番に実行」「枝分かれ」が学べるという主張は納得できますか? ちょっと違うと思いますか? 違うとすればどのように?
- 「枝分かれが増えるとどんどんプログラムは複雑になる」という表明については納得が行きますか? 複雑になるとしたら、その複雑さにどう対応したらいいと思いますか?
- インタラクション (プログラムとのやりとり) があると、これまでとだいぶ違うことができる、という説明は納得しましたか。
- ゲームの難しさは人によって違ったり、バランスが大事だったりという話は納得しましたか。

- 指数的増加/線形増加やシミュレーションの価値については納得が行きましたか。
- 二進法をビスケットでプログラムするのを見てどう思いましたか。二進法と十進法ではかなり複雑さが違うという話についてはどうでしたか。
- オルゴールのプログラムを見て「読める」と感じましたか。プログラムを読めることが大切という説明は納得しましたか。

# 第3章　ドリトルによるプログラミング学習

ドリトル (Dolittle) は兼宗　進先生 (現: 大阪電気通信大学) が筑波大学社会人大学院時代に筆者 (久野) と共同で開発した教育用プログラミング言語ならびに環境です。ドリトルは文字の並びでプログラムを表現する (伝統的な) テキスト型のプログラミング言語ですが、日本語の文字や名前を用いて記述し、初心者のつまづきを減らすような工夫を多く盛り込んでいます。また、教育向け言語の定番だった LOGO が持つタートルグラフィクス機能を取り込み、今日的なオブジェクト指向の枠組みで多様な部品を活用できるようにしています。本章ではテキスト型、変数、名前など、ビスケットには含まれていなかったプログラミング言語の特徴を扱うプログラミング学習について取り上げます。

## 3.1　テキスト型プログラミング言語

プログラミングには通常、「プログラミング言語」を使用します。ここで「言語」と呼ばれるのは、「文字の連なりでさまざまな意味 (動作の指示) をコンピュータに伝えている」ことによります。このような、文字の連なりで記述するプログラミング言語を「テキスト型」プログラミング言語と呼ぶこともあります (教育向け以外の分野では、大半のプログラミング言語はテキスト型です)。

テキスト型でないものとしては、スクラッチのようにブロックのはめ込みによって指示を組み立てたり、流れ図で指示を組み立てたりするものがあります。このような、視覚的な配置を用いるものは「ビジュアル」プログラミング言語と呼びます。ビスケットも「めがね内の絵の配置」によって動作を指示するので、この中に入るでしょう。

さて、ではなぜ世の中にはテキスト型言語が多いのに、教育用では必ずしもそうなっていないのでしょうか。それには次の 2 つの理由があります。

- 文字の並びとしては日本語や英語などの語彙を多く使うので、これらの語彙を扱える学齢にならないと学びにくい。
- 書き方の規則が厳密であり、そこから外れているとエラーとなって動作させられないので初心者にはハードルが高い。

だったらなぜここでテキスト型を扱っているのか、ということになりそうですが、上記の半面、テキスト型には次の特徴があります。

- 記述がコンパクトで緻密であり、「名前」によって遠くの (今画面に見えていない) 対象も容易に指示できるので複雑なプログラムが構成できる。[1]
- テキストであるので、キーボードを使ってテキストエディタで入力や修正ができ操作の効率が良く、またさまざまなツールで柔軟に管理できる。

これらの利点は大規模で複雑なプログラムを作るときには欠かせないものであり、その一方で先に挙げた「弱点」は熟練した大人にとってはさして問題になはりません。このため、世の中のプログラミング言語の多くはテキスト型なわけです。そして教育用途の場合も、どこかから先は (キーボードから文字が打て、書き方の規則を理解して扱えるようになっていれば)、テキスト型でも大丈夫です (教え方にはそれなりの配慮は必要でしょうけれど)。

[1] もちろんビジュアル型の言語でも手続きや変数に名前をつけたりしますが、これはその部分だけテキスト型のツールを借りてきていると解釈することもできます。

そういうわけで、本章で示すドリトルを用いた授業案では、(もちろん)1 章で取り上げたプログラミング学習の基本方針に沿うことに加えて、上の 4 つの点を気付いてもらうことにも重点を置いています。また、ビスケットの授業案が小学校 1 年生から使えることを想定していたのに対し、本章では小学校高学年以上を想定し、その発達段階で扱えるような概念について取り上げて行きます。

## 3.2 基本動作: タートルで線画を描く

### 3.2.1 概要と授業案

ドリトルは前述のようにテキスト型のプログラミング言語ですが、それに加えて、LOGO 言語以来教育向けの素材として定評のあるタートルグラフィクスと、今日の汎用プログラミング言語においては欠かすことのできないオブジェクト指向の概念を取り入れています。初回の授業案から、これらの点はすべて現れています (表 3.1)。

表 3.1: ドリトル 1 時間目: タートルで線画を描く

| | |
|---|---|
| 導入 (5 分) | ドリトルはテキスト型 (文字で記述するタイプ) のプログラミング言語であることを説明。書き方は厳密に決まっていることを説明。名前は「既に用意されている名前」と「自分でつけていい名前」があることを説明。 |
| 最初のプログラム (10 分) | ドリトルを起動し、編集タブに次のプログラムを打ち込む (図 3.1)。<br>　　かめた＝タートル！作る。<br>タートル (英語で「カメ」) とは「画面に絵を描くためのペン先」。「！」は以下の命令をするというしるし。「作る」は新しいもの (この場合はタートル) を作るという命令。「かめた」はとりあえず 1 つ作ったタートルにつけた名前。「かめた＝…」は作ったものに名前をつけている。「。」は 1 まとまりの動作の終わりのしるし。この状態で実行ボタンを押し、画面にタートルが現れるのを見せる (図 3.2)。「！」が無かったり「。」が無かったり「タートル」「作る」が違うとエラーになることを見せ (図 3.3)、間違えたらエラーが出るけど直せばいいからと説明。「かめた」は自分でつけた名前だから「かめこ」でも動くと説明。ここまで全員にやってもらう。わざと間違えてみてエラーも体験するように言う。 |
| 線画の機能の説明 (5 分) | 　　かめた！100　歩く。<br>を次の行として追加し、線が引けることを見せる (空白は必要と説明)。<br>　　かめた！90　右回り。<br>も見せ、向きが変わることを見せる (左回りもあるという)。 |
| 線画を描く (20 分) | まず同じ例を試してから、ノートに一筆描きでかける簡単な (直線だけの) 線画を描いて「設計」し、それをかめたに描かせてみるように指示。三角形やジグザグを黒板に描いて例示。途中の適当な時点で<br>　　かめた！100　歩く　90　右回り　100 歩く。<br>のようにつなげて書いても大丈夫なことを説明し、またタートルの命令一覧 (表 3.2) を配布。 |
| まとめ (5 分) | 気がついたことや感想を自由に言ってもらう。 |

この授業案の前半部分は、テキスト型の言語で現れる「約束ごと」をドリトルの具体例に即して説明することが中心になっています。そのために「かめた＝タートル！作る。」という最初の文だけでかなり丁寧に説明し、また実際に体験させるようにしています。とくに、「間違った書き方でエラーがでること」を普通のこととして例示し、体験させるようにしていることが特徴です。

この段階で出て来るドリトルの構文 (書き方) は次の 2 つです。

変数 ＝ オブジェクト！ パラメタ… 命令 パラメタ… 命令 … 命令。
オブジェクト！ パラメタ… 命令 パラメタ… 命令 … 命令。

1 番目と 2 番目の違いは変数に値を入れるかどうかの違いだけです。変数は「値を入れるいれもの」であり、その名前はプログラムの作成者が自由に選べます。なお、名前は「漢字、かな、英数

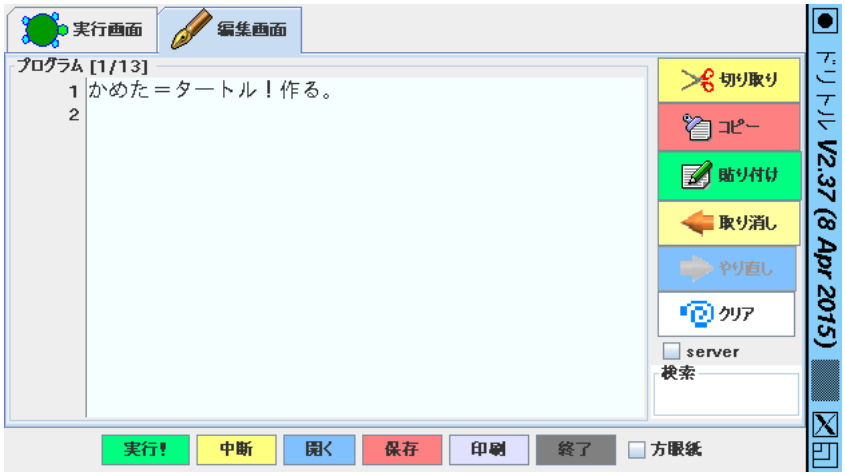


図 3.1: ドリトルのプログラムを入力しているところ

字のならびで、ただし数字からは始まらないようなもの」としています。いちど値を入れた変数は、その入れた値を繰り返し使うために利用できます。

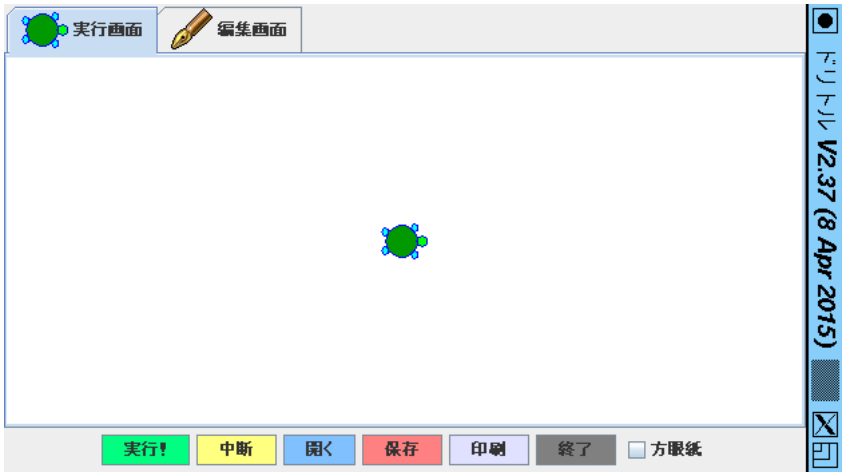


図 3.2: 最初のプログラムを実行したところ

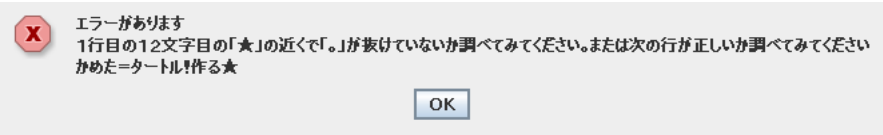


図 3.3: 「。」を入れ損なった場合

この時間の例では「かめた」という名前の変数に作ったタートルを入れて、その後それを繰り返し参照しています。実習時には学習者にも自由に名前をつけさせるとよいでしょう。名前を変えた場合、後から参照するときもその同じ名前にする必要があることに注意させてください。

さて「オブジェクト」ですが、これはドリトルの場合「値」と同じ意味になります。ただ、値と呼ぶと数値などが思い浮かびますが、タートルなど複雑な機能を持つものも同じように値として扱うので、「複雑な機能を持った値」という意味を込めてオブジェクトと呼んでいます。オブジェクトとしては、ドリトルの方で予め用意している名前を使う場合も、自分で決めた変数を使う場合もあります。

そして、オブジェクトの次に「！」を置き、その後にオブジェクトに対する命令 (ドリトルの用語ではメソッド) を複数書けます (1 つだけでも構いません)。どのような命令が書けるかは、オブジェクトの種類ごとに決まっています。今回はタートルオブジェクトだけが出て来るので、書け

る命令は表 3.2 にあるものになります。

命令には (どれだけ前に進むかなどの) 付属した値を指定できますが、これをパラメタと言います。数値のパラメタはそのまま直接書けますが、変数や変数を含む計算式は「(…)」で囲んだ形で書く必要があります (そうしないと命令と形の上で区別がつかないため)。

だいぶ細かい説明になりましたが、これらの説明はもちろん詳しくする必要はなく、ただ尋ねられたときに備えていちおいう理解しておいて頂けば充分でしょう。

授業の後半部分では、「歩く」「右 (左) 回り」の命令を例示し、これらを使えばさまざまな線画が描けると分かってもらった上で、あとは自由に作品を計画してもらい、制作させることが内容となっています。

タートルグラフィクスは、このように簡単な命令だけでかなり「思ったものを作る」ことを可能にしてくれるという点で価値があります。しばらくやったら、もっと別の命令が知りたくなると思うので、頃会いを見計らって表 3.2 を紹介します。

表 3.2: タートルのメソッド

| 名前 | 機能 | サンプル |
|---|---|---|
| 作る | 新しいタートルを作る | かめた＝タートル！作る。 |
| 歩く | 前に進む | かめた！１００　歩く。 |
| 戻る | 後ろに戻る | かめた！１００　戻る。 |
| 右回り | 右に指定角度だけ回る | かめた！９０　右回り。 |
| 左回り | 左に指定角度だけ回る | かめた！９０　左回り。 |
| 移動する | 右に $x$、上に $y$ 移動 | かめた！１０　２０　移動する。 |
| 位置 | 指定した位置 $(x, y)$ に移動 | かめた！５０　－５０　位置。 |
| ペンあり | 線を引くようになる | かめた！ペンあり。 |
| ペンなし | 線を引かないようになる | かめた！ペンなし。 |
| 中心に戻る | 画面の中心に移動 | かめた！中心に戻る。 |
| 閉じる | 描き始めの点に移動 | かめた！５０　歩く　３０<br>右回り　５０　歩く　閉じる。 |
| 線の色 | 線の色を変更する | かめた！(緑) 線の色。 |
| 線の太さ | 線の太さを変更する | かめた！５　線の太さ。 |
| 図形を作る | タートルの歩いたあとから図形を作る | 四角＝かめた！図形を作る。 |

### 3.2.2　何を気付いてもらうか

本章でも各回の授業内容を実施した時に気付いてもらえる事柄について挙げておきます。前章と重複する内容もとくに除外はしていませんが、テキスト型言語に関する部分は当然新しい内容となります。

- プログラミング言語の記述は「日本語」ではない — 分かりやすさのために日本語の単語は使っていますが、プログラミング言語はあくまでもコンピュータに対する指示を記述するための人工的な「書き方の規則」であり、日本語として読めるからその通り実行できる、というわけではありません。

- わかち書きのための空白や「！」「。」などの記号が適切に入っていないと構文エラーとなる — 日本語などでは句読点類は無くても意味が通りますし空白はそもそも不要ですが、プログラミング言語は厳密な規則に従っていて、これらが無いと正しく認識されません。
- 構文エラーが 1 箇所でもあると実行されない — プログラミング言語の実行系はこうなっていることが多いのですが、これは日本語などのように「分からないところは無視しても残りは役に立つ」のと大きく違うのでとまどいやすいところです。
- 既存の名前 (オブジェクトやメソッドの名前) が間違っているとその名前を利用したあたりでエラーで止まる — 最初から実行がはじまらない構文エラーとは別に、実行開始した後で止まるような種類のエラーも多く存在しています。
- 同じ動作のプログラムでも何通りにも書ける — 命令を複数まとめて書くようにもできるなど、プログラムの書き方には自由度があります。空白の入れ方や改行の入れ方なども自由度があります。プログラミングでは、このような自由度の中で、自分なりの「わかりやすい/きれいな」書き方を考えて身につけることが求められます。
- プログラムは「順番に」実行される — まず「かめた」が作られ、次に 100 だけ線を引き、…のようにプログラムは書かれた順番に (横書きの文章と同様に) 上から下、左から右に実行されて行きます。このこと一見は当り前のようですが、順番が厳密に保証されることで込み入った記述ができるという面もあります。このことに気付かせるには、プログラムが数行になったとき「この行とこの行を入れ替えるとどうなるかな」などと問いかけてみる方法もあります。
- 非常に込み入った動作や手順であっても、プログラムを変更しなければ、何回やっても同じ結果となる — 日常生活では込み入った作業は実行するたびに違いが生まれるのが普通ですが、プログラムの場合は「さいころを振る (擬似乱数)」動作を使わない限り (または値が変化する外部入力がない限り)、どんなに複雑な動作でもまったく同じに実行されます。

ここに挙げた点はいずれも、日常生活とは極めてかけ離れていますが、コンピュータとプログラミングの世界では「一般的で当り前なこと」です。これらについて意識しておくことは、コンピュータや情報システムとつき合う上で有用だと考えられます。

## 3.3 繰り返しの威力/変数の威力

### 3.3.1 概要と授業案

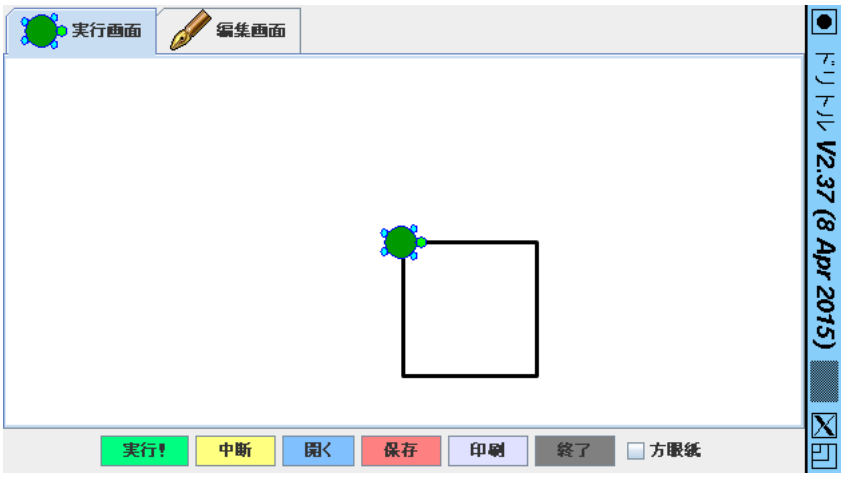


図 3.4: 繰り返しを用いて正方形を描く

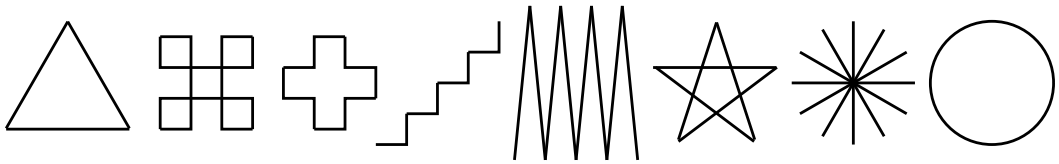

図 3.5: 繰り返しで描ける図の例

前節で出て来たプログラムでは、個々の命令は1回ずつしか実行されないので、プログラムの長さに相応した線しか描けませんでした (このことは前節における重要な「気付き」に含めてもよいでしょう)。これに対し「繰り返し」の機能があると、一部のプログラム片 (コード) が何回も実行されるため、少量のコードでも沢山の仕事をさせられます。また、繰り返しの中にある個々の命令は単純なものですが、そこに「どれだけ」という「違い」を (変数の導入によって) 追加すると、多くの複雑な動作が行わせられます。では、これらを扱う授業案を見てみましょう (表 3.3)。

表 3.3: ドリトル1時間目: タートルで線画を描く

| | |
|---|---|
| 導入 (5 分) | プログラムでは「繰り返し」により少量のコードからコンピュータに大量の仕事を行わせられることを述べる。 |
| 繰り返しのあるプログラム (15 分) | 正方形を黒板に描き、どんなプログラムで描くか問いかけた後、次のプログラムを見せる (図 3.4)。<br>かめた＝タートル！作る。<br>「かめた！100 歩く 90 右回り。」！4 繰り返す。<br>同じ動作を沢山やるには「」で囲み「繰り返す」を使うと説明。繰り返しを使って描ける絵の例 (図 3.5) を示し、自由に作ってみるように指示。 |
| 変数の説明 (5 分) | うずまき (図 3.6) をプログラムを隠して見せ。これはどのようにしたら描けると思うか考えさせる。一定量ずつ線が長くなっていることに気付かせる (または指摘する)。プログラムを見せる。<br>かめた＝タートル！作る。<br>長さ＝１０。<br>「かめた！(長さ) 歩く 90 右回り。長さ＝長さ＋ 10。」！20 繰り返す。<br>「長さ＝ 10。」で長さという変数に最初の値を入れ、「長さ＝長さ＋ 10。」でこれまでの長さに 10 足したものを新たな長さとできることを説明。オブジェクトの命令のパラメタに変数を使うときは「(長さ)」のようにかっこで囲むことも説明。 |
| 変数を用いた作画 (20 分) | 自由に変数を用いた作画を行ってもらう。適当なところで例として図 3.7 なども見せて描き方を考えてみてもらう。 |
| まとめ (5 分) | 気がついたことや感想を自由に言ってもらう。 |

なお、数値に対しては表 3.4 のように、四則演算 (＋, －, ＊, ／) と剰余 (％) が使えます。また先に述べた「乱数」もあります。あと、次節で条件に大小比較を使うのでそれも掲載します (「等しい」はイコール記号 2 つなのに注意)。比較演算の結果は「はい」「いいえ」のいずれかの値 (真偽値) です。このほかに三角関数なども使えますがここでは略します。

繰り返しと変数の両方を扱うというのはかなり欲張っていると言えます。状況に応じて 2 回に分けて実施することも考えられます。

### 3.3.2 何を気づいてもらうか

本節の授業案は繰り返しと変数の 2 つを含むという欲張った内容ですが、両者の組み合わせによって分かることもかなり多いと考えます。以下では繰り返しに関するところと変数に関するところを分けて挙げます。繰り返しに関しては次のものがあるでしょう。

- 繰り返によって多くの仕事が行わせられる — 短いプログラムであっても、その一部を繰り返すことで多数の命令を実行でき、多くの仕事を行わせられます。つまり役に立つプログラ

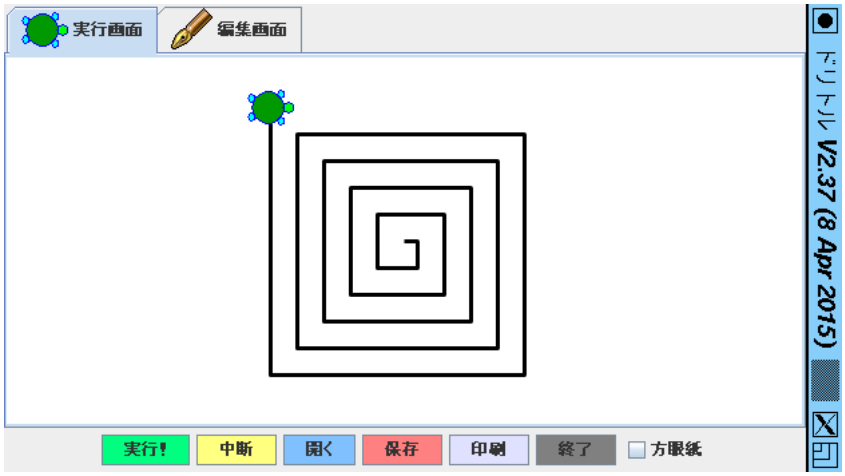


図 3.6: うずまき

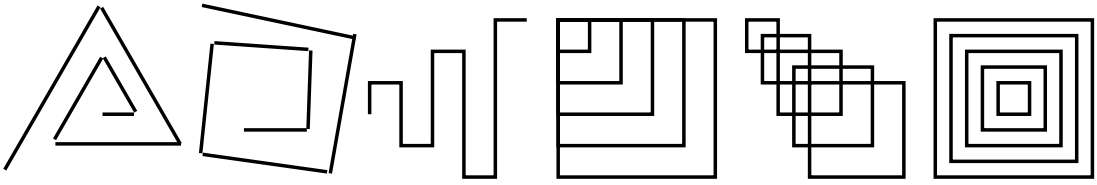

図 3.7: 変数と繰り返しの併用で描ける図の例

表 3.4: 数値の演算子とメソッド

| 名前 | 機能 | サンプル |
|---|---|---|
| ＋ | 加算 | ｘ＋１ |
| - | 減算 | ｘ－ １ |
| ＊ | 乗算 | ｘ＊２ |
| ／, ÷ | 除算 | ｘ／２, ｘ÷２ |
| ％ | 剰余 | ｘ％２ |
| 乱数 | その値未満の整数の乱数を返す | １００！乱数 |
| ＞ | より大 | ｘ＞０ |
| >=, ≧ | 以上 | ｘ≧０ |
| ＜ | より小 | ｘ＜０ |
| <=, ≦ | 以下 | ｘ≦０ |
| ＝＝ | 等しい | ｘ＝＝０ |
| !=, ≠ | 等しくない | ｘ≠０ |

ムが作れます。

- コンピュータの処理は非常に高速である — かなり大量の線を含む複雑な図形であっても、あっという間に処理が終わります。どちらかというと、計算した後、その計算に従って画面を描くのが時間が掛かります (観察すれば分かると思いますが)。
- コンピュータによる計算はかなり正確である — コンピュータによる数値計算は (通常のやり方では)16 桁程度の制度ですが、これは近似値ではあってもかなり高い精度です。これにより、たとえば 360 角形を描くことで「円」を描いても、ぴったり閉じた円ができるわけです。人間が同じことをやろうとしても、きちんと最初の位置に戻ることは困難でしょう。
- 計算結果は何回繰り返しても同じ値が再現される — たとえば正方形を描くプログラムで回数を 100 とかにしても、見える図形は変わりません。これは同じ正方形の上を何回ぴったりなぞっても絵は同じだからです。人間が繰り返しなぞるとぶれたりして線が太くなりますが、コンピュータではそういうことにはならないわけです。

このように、繰り返しは「同じことをやる」わけで、場合によっては同じことを何回もやってもらってもそれが欲しいものではない、ということが起きます。ここで、「変数」と「値の計算」を入れることで、次のようなことが可能になります。

- 変数は「どれだけ」という値を覚えておくことや、それを規則的に変化さていくという機能を提供する — 繰り返しの中で変数を使って線の長さなどを変化させて行くことで、徐々に大きく/ 小さくなるような図形を描くことができます。
- プログラムでは「状態」の概念が重要 — プログラムは「状態」が同じであればまったく同一に動作します。状態にはまず、プログラム上の「どの箇所を実行しているか」があります。次に、前節までのプログラムでは「タートルの位置や角度」があります。これだけだと「正方形を何回もなぞっても絵は同じ」わけですが、今節でこれに「変数の値」が加わることでより複雑な状態が持てるようになったわけです。

ところで、プログラムで「毎回同じ結果」であって欲しくないときには、乱数を使うという話が出ていました。具体的には「$N$！乱数」という命令を使うことで、0〜$N-1$の値から「サイコロを振って 1 つ」選ぶことができるのです。たとえば次のプログラムを見てみましょう。

```
かめた＝タートル！作る。
「かめた！（360！乱数）右回り 20 歩く。」！200 繰り返す。
```

このプログラムでは、20 だけ歩くごとにサイコロを振って歩く向きを 0〜359 度の範囲でランダムに選ぶので、酔っ払いの千鳥足のような「ランダムウォーク」の絵が描けます。ただし、この場合でも状態が無いわけではなく、0〜$N-1$の間から 1 つ次の値 (=状態) を選んでいることに注意しましょう。

なお、ここで使っている乱数の機能は正確には「擬似乱数」と呼ばれます。というのは、コンピュータでは計算の動作は厳密に決まっているので、本当に「ランダム」にはできないからです。代わりに、複雑で結果がランダムに「見える」計算式を使って順番に値を計算しています。そのため「擬似 (本ものではない)」乱数、なわけです。

## 3.4 入力・出力と GUI 部品

### 3.4.1 概要と授業案

前節までのプログラムはすべて、実行を開始すると絵が出力されておしまい、というものであり、プログラムの実行中にユーザとやりとりするという機能は実現されていませんでした (このことは「気付き」に含めてもよいと思います)。これに対し、私たちが普段目にするプログラム (ソ

フトウェア) は、ユーザが操作するとそれに応じて応答する、という対話的な機能を持っているということが大きく違っています。

実はここまで動かして来たプログラムも、入力が無いわけではなく、プログラムの一部として (線の長さや描き方などを) 予め書き込んであって、それによって求める絵が作れるわけです。しかし、そのようなプログラムでは、プログラムを作る人だけがプログラムの動きを調整できますから、誰にでも使ってもらうことは難しいといえます。これに対し、プログラム実行開始後に入力を受け付けるプログラムであれば、さまざまな人に使ってもらうことが可能であり、プログラミングの体験も豊かなものとできます。


図 3.8: 摂氏華氏変換

ところで、従来型のプログラミング環境では、GUI を扱うまでのハードルが高いので、CUI(コマンド行型) の入出力を用いて演習するのが普通です。しかし、「普段目にするプログラム」における入出力は GUI が多いわけです。ドリトルはオブジェクト指向 (ある程度複雑な機能を持ったまとまりをオブジェクトといい、タートルもその 1 つです) を活かして、GUI も容易に扱えるように準備されています。たとえば、図 3.8 は次の授業案で扱う最初のプログラムの画面です。授業案は表 3.5 にあります。

一般に、プログラムは「順次実行 (連接)」「反復」「分岐」を組み合わせてさまざまな処理の流れを作って行きます。これを一般に制御構造と言います。先に連接と反復が出てきていたので、この回の「分岐」を合わせることで制御構造がひととおり学べます。ドリトルでは反復や分岐はブロックのメソッドとして実現されていて、他の言語とは少し書き方が違っています。主要なものを整理しておきます。繰り返しには、既に学んだ回数指定のもののほかに、条件指定のものもあります。

「…繰り返す動作…」!N 繰り返す。

「…条件…」! の間「…繰り返す動作…」実行。

枝分かれについては、例題にあった「条件が成り立つ時だけ実行」のほかに、「それ以外の場合に実行」する動作を指定するものもあります。また、それ以外の場合に再び条件を指定して枝分かれすることもできます。いずれも、最後の「実行。」を忘れやすいので注意が必要です。

「…条件…」! なら「…条件成立時の動作…」実行。

「…条件…」! なら「…成立時の動作…」
そうでなければ「…不成立時の動作…」実行。

表 3.5: ドリトル 3 時間目: GUI による入出力/制御構造

| | |
|---|---|
| 導入 (5 分) | プログラムの実行途中でユーザとのやりとりが行われることを取り上げ、スマホアプリなどでの具体例を挙げてもらう。 |
| GUI を使った入出力 (20 分) | 入力欄＝フィールド！作る。<br>計算ボタン＝ボタン！ "計算" 作る。<br>表示欄＝ラベル！ "華氏から摂氏に変換します" 作る。<br>の 3 行で動かし、部品が表示される様子を見せる。作ってみるように指示。ボタンを押しても何も起きないことを指摘。次にボタンの動作を指定。<br>計算ボタン：動作＝「<br>x ＝入力欄！読む。<br>y ＝ (x - 32)*(5 / 9)。<br>表示欄！("摂氏の温度："＋ y) 書く。」。<br>ここで最後の『」』は動作の次の『「』と対応していることを指摘。まずこれと同じものを動かし、続いてさまざまな計算式の計算をしてみる時間を取る。 |
| 枝分かれの使用 (5 分) | ans ＝ 100！乱数。<br>計算ボタン：動作＝「<br>x ＝入力欄！読む。<br>「x ＞ ans」！なら「表示欄！"大きい" 書く。」実行。<br>「x ＜ ans」！なら「表示欄！"小さい" 書く。」実行。<br>「x == ans」！なら「表示欄！"正解！" 書く。」実行。」。<br>このように条件が成り立った時だけ実行する動作を指定することで、条件に応じた処理が記述できるようになる。実際に数当てを動かし、改良したいと思うことを述べさせる。 |
| 条件を持つプログラム (15 分) | 先のプログラムに条件を盛り込んだものを作ってもらう。または、数あてプログラムの改良を考え実装する。 |
| まとめ (5 分) | 動作の順次実行と繰り返しと枝分かれを合わせて「制御構造」と呼ぶ。気がついたことや感想を自由に言ってもらう。 |

「…条件 1 …」！なら「…条件 1 成立時の動作…」
そうでなければ「条件 2 」なら「…条件 2 成立時の動作…」
そうでなければ「条件 3 」なら「…条件 3 成立時の動作…」
そうでなければ「…いずれも不成立時の動作…」実行。

表 3.6: さまざまな値の計算式

| | |
|---|---|
| 温度変換 (華氏 F →摂氏 C) | $C = (F - 32) \times \frac{5}{9}$ |
| 温度変換 (摂氏 C →華氏 F) | $F = C \times \frac{9}{5} + 32$ |
| 伸長 T、体重 W に対する BMI | $BMI = \frac{W}{T \times T}$ |

また、この案では数値に対する処理ばかりの内容となっていますが、入力された数値を長さや角度として扱い、タートルで絵を描かせることもできます。そのような体験も含めるのであれば、この回の授業案も前半 (入出力) と後半 (枝分かれ、制御構造) に分けて実施するのがよいでしょう。

### 3.4.2 何を気づいてもらうか

本節の授業案では入出力 (GUI) と制御構造が重要なポイントとなります。それぞれについて分けて、気づいてもらいたいことを挙げます。まず入出力からです。

- 入力に対してどのような結果が返されるかは、入力の値と入力時点でのプログラムの状態によって決まる — プログラムの状態は、前節でやったように、変数の値 (とタートルなどのオブジェクトの状態) から成ります。コード上の実行箇所は、ボタンを押すことによって計算を実行するのであれば、いつも同じになります (ボタンを 2 つ以上にしてそれぞれで別の計算をすればもちろんそれらの間で異ります)。
- GUI のプログラムはイベントドリブン — ここで扱っている GUI による入出力では、入力欄の値はユーザの操作でいつでも変化させられますが、処理そのものはボタンを押したタイミングで動作します。このように、外部のできごと (この場合はボタン押し) に対応して処理が行われるスタイルのプログラムをイベントドリブンと言います。
- 出力の形に複数の枠組みがある — GUI を用いたプログラムでは、GUI 部品 (今回はラベル) の中に文字列を書くことでそれが表示される、という形の出力が多く使われます。しかしそれとは別に、タートルグラフィクスも使うことができます。つまり、入力がイベントドリブンに絞られているのに対し、出力はさまざまな枠組みが使えるといえるでしょう。

次に、制御構造についてです。

- 枝分かれは多くの実行経路を作り出す — 枝分かれの命令があると、条件に応じて「ある処理を行う」「行わない」の 2 通りの経路ができます (行わない別の処理を行ったり、複数の処理から 1 つが選ばれるなどもありますが、原理的には同じことです)。すると、枝別れが 2 つあれば $2^2 = 4$、3 つあれば $2^3 = 8$、というふうにプログラムに多くの実行経路が生まれ、処理が複雑になります (ということは、プログラムのふるまいも複雑になるわけです)。
- 制御構造は組み合わせて使う — 繰り返しの中で枝別れとか、枝別れの中で繰り返しとか、制御構造は複数のものを組み合わせていくことでより込み入った構造を作り出すことができます。
- 一般的にどう動くかを考える必要性 — 上に書いたように、制御構造が組み合わさるとプログラムの動作が複雑になるので、やみくもにコードを書いているとどのような動作か分からなくなります。これを克服するには「この部分で一般に (さまざまなデータが来た場合でも) どのような変化が達成されているか」を考えて行くことで、心理的な「場合の数」を減らすことが重要になります。

## 3.5 図形/タイマー/ゲーム

### 3.5.1 概要と授業案

ドリトルの 4 時間目はある程度ゲーム的なものをテーマとしています。これは 1 つには、児童・生徒の中にプログラミングに対するモチベーションとして「自分でゲームを作る」ことを挙げる者がある程度いると思われるからですし、もう 1 つはゲームを扱うことで「人間の特性」「ユーザの気持ち」に目が向くという面があるからです。

授業案を図 3.7 に示します。ゲームにする前にまず、これまでにタートルグラフィクスで描いてきた絵を「図形オブジェクト」にできる、という話があります。前半の例ではまずタートルの軌跡から「四角」を作り、次にそれをコピーした「四角 2」を作っています (図 3.9)。

なお、細かい話ですが、「図形を作る」という命令はタートルに対する命令で、そこから返ってくる値は図形 (この場合は正方形) ですので、次の命令「塗る」はこの図形に対して送られます。図形に対して使える命令の一覧は表 3.8 にあります。

表 3.7: ドリトル 4 時間目: 図形/タイマー/ゲーム

| | |
|---|---|
| 導入 (5 分) | 「ゲームって何?」と問いかけ、どのようなものであればゲームと言えるのか考えてもらう。画面にものが現れてユーザの操作に応じて変化したり動くようなプログラムを作ると予告。 |
| 図形オブジェクト (15 分) | 四角を 2 つ描くといって図 3.9 を見せる。<br>かめた＝タートル！作る。<br>「かめた！100 歩く 90 右回り。」！4 繰り返す。<br>で描いた四角を図形にできるという説明をし、次の 2 行を加える。<br>四角＝かめた！図形を作る（緑）塗る。<br>四角 2 ＝四角！作る（青）塗る 50 50 位置。<br>作るというのはコピーを作るという意味になると説明。位置をはじめ図形の命令は表 3.8 を見せる。自分でも図形を作ってもらう。 |
| タイマーと衝突 (5 分) | 図 3.10 のプログラムを示し、説明する。四角を 8 個コピーして画面に配置している (最初の四角は使わないので消す)。ボタンを作り、それぞれのボタンでかめたを回転させる。最後にタイマーは一定時間間隔で「…」の動作を (この場合は 30 秒間) 繰り返し実行させる。ゲームを動作させる。 |
| 自由制作 (15 分) | 先のプログラムを作ってもらう。改良したいところを言ってもらい、改良してもらう。 |
| まとめ (5 分) | ゲームとは結局なんだったか? |

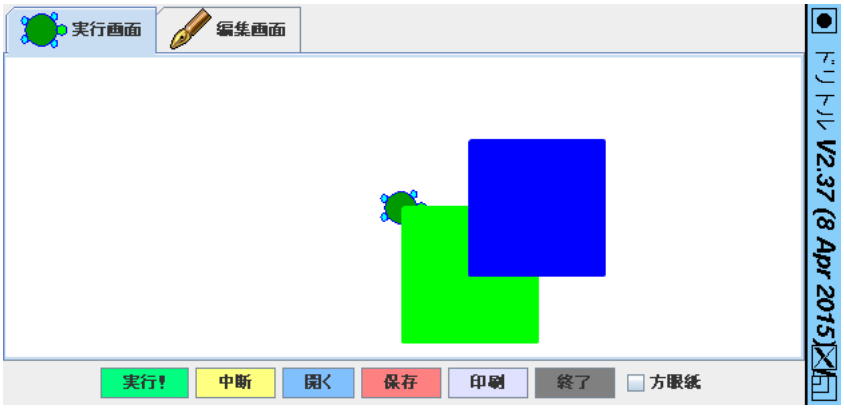


図 3.9: 四角を 2 つ表示したようす

表 3.8: 図形のメソッド

| 名前 | 機能 | サンプル |
|---|---|---|
| 右回り | 右に指定角度だけ回る | 四角！３０　右回り。 |
| 左回り | 左に指定角度だけ回る | 四角！３０　左回り。 |
| 移動する | 右に $x$、上に $y$ 移動 | 四角！１０　２０　移動する。 |
| 位置 | 指定した位置 $(x, y)$ に移動 | 四角！５０　－５０　位置。 |
| 塗る | 色を塗る | 四角！(赤）塗る。 |
| 拡大する | 指定倍率に拡大/縮小。 | 四角！1.5 拡大する。 |
| 消える | 画面から消える | 四角！消える。 |
| 現れる | 画面に現れる | 四角！現れる。 |

次に後半では、図 3.10 に掲載したゲームのプログラムが題材になっています (実行のようすは図 3.11)。タートルが動いていって図形に衝突すると「衝突」で定義されたブロックが動作し、そこでブロックに対して衝突した相手の図形が渡されるので、これに対して「消える」命令を実行します。

タートルそのものは、2 つボタンを作ってこれらで左右に回転できるようにしています。タイマーは「短い間隔で繰り返し動作を実行する」機能を提供し、ここでは 30 秒間かめたを 5 歩ずつ歩かせることで徐々に前進させているわけです。

```
かめた＝タートル！作る。
「かめた！40 歩く 90 右回り」！ 4 繰り返す。
四角＝かめた！図形を作る （緑） 塗る。
ｘ＝ － 250。
「四角！作る （ｘ） （- 60） 位置。 ｘ＝ｘ+60。」！ 8 繰り返す。
四角！消える。
かめた:衝突＝「| ｚ | ｚ!消える。」。
右ボタン＝ボタン!"右" 作る。右ボタン:動作＝「かめた!10 右回り。」。
左ボタン＝ボタン!"左" 作る。左ボタン:動作＝「かめた!10 左回り。」。
タイマー!作る 30 時間「かめた！ 5 歩く」実行。
```

図 3.10: 四角を全部消すゲーム

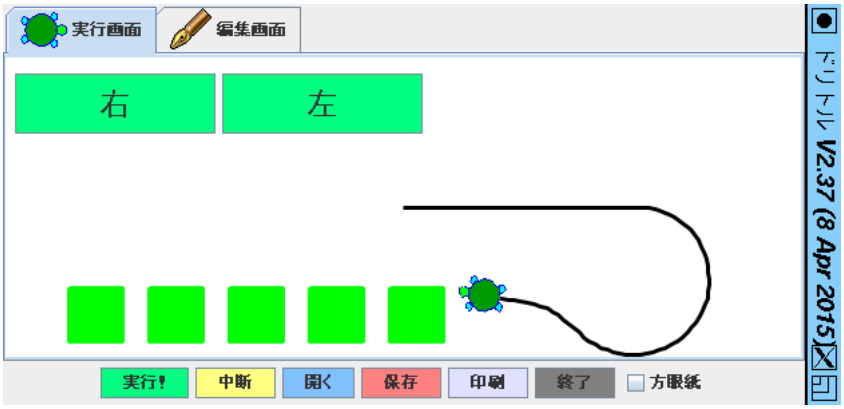


図 3.11: 四角を消すゲームのようす

このゲームに対する改良としては、消しおわった四角の数を表示する、四角を全部消しおわったら成功メッセージを表示する、時間切れになったらゲームオーバーと表示する、などがあるでしょう。なお、タイマーの動作が完了するまで待つには「待つ」命令を実行すればよく、この命令が終わったあとに個数が全部でなければゲームオーバーのメッセージを表示する、などとすればよいでしょう。なお、例題そのものは前節で学んだ枝分かれを使っていませんが、上に挙げたような機能を付加しようとしたときには当然、枝分かれが必要となります。

### 3.5.2 何を気づいてもらうか

この節ではタイマーによるアニメーションを取り入れることで、実時間性のある (応答の内容に加えてタイミングも問題になる) ゲームを構成しています。この節で気づいて欲しいことの半分は、このようなプログラムの特性に関することですが、残りの半分はこのようなプログラムを通じて分かる人間の特性に関することになります。まずプログラムの特性に関する部分から示します。

- アニメーションの原理 — アニメーションは一定時間間隔で絵を変化させることで実現できます。今回の場合はタートルを少しずつ前進させるという変化ですが、これで「タートルの位置と軌跡だけが少し変化し、あとは前と同じまま」という絵が次々に作られるわけです。
- ユーザによる対話的アニメーションの原理 — ユーザ入力は前記のアニメーションによる「変化のさせ方」を変化させることをおこないます。つまり、ユーザ入力は直接絵を操作するのでなく、次々に生成される絵の「変化のしかた」(ここではタートルの向き) を制御しているわけです。
- 画面上のものどうしの作用 — ユーザ入力がイベントドリブンという話は先に出てきましたが、今回はタートルが四角に接触すると消える部分がイベントドリブンで書かれています。画面はすべてコンピュータが描いているのですが、アニメーションの場合その絵が刻々と変化するので、絵どうしの接触などを (外部入力とは関係のない、擬似的な) イベントとして扱う方がプログラムが分かりやすいのです。

次に人間の特性に関する部分です。

- 粗い変化でも人間には連続的に見える — タイマーによる絵の変化の時間間隔は (とくに変更しない限り)1 秒間に 10 回程度ですが、それでも充分絵が動いて見えます。人間の時間に対する感覚はその程度のものだということです。
- 間接的な操作の難しさ — 左右のボタンを押すとタートルの向きが変えられて操縦できるというのはすぐ分かりますが、思った通り操縦するのはなかなか難しいです。これは、ボタン操作→タートルの左右の回転→ 1 ステップずつ進む向きの変化、というつながりが間接的で「すぐに」思ったようにはならない時間的な遅れが介在しているためです。
- 難しさの調節が可能 — タートルが 1 回に進む長さやボタン 1 回で回転する角度を調節することで、ゲームの難しさを調節できます。これは前述の時間遅れの効果を減らしたり増やしたりしているものと考えられます。そして、どれくらいがちょうどいい難しさかは、人によってさまざまです。
- 人間とコンピュータの対話モデル — ここまでのところを捉え直すと、コンピュータと人間のやりとりは「人間の操作→コンピュータの処理→画面への反映→人間による結果の知覚→人間による操作…」というサイクルが繰り返されてできていると分かります。実時間でないプログラムはこのサイクルをゆっくり (人間のペースで) 回せるのに対し、実時間だとそうではないという違いがあるわけです。

## 3.6 ドリトルのまとめ

ドリトルをビスケットと対比させた場合のいちばんの違いは、やはりテキスト型の言語という点になるでしょう。本章冒頭でも述べましたが、テキスト型はコンパクトであり、名前により多くの対象が簡単に指し示せ、キーボードで入力・修正できるという利点があります。世の中の実用言語はほぼすべてがテキスト型なので、それに慣れるのはさらに学習を進めたい人にとってはよいことでしょう。

次に、他のテキスト型言語と比べた場合の特徴も整理しておきます。

- タートルグラフィクスにより、プログラムとその出力の関係が把握しやすく、「思ったものを作る」体験が達成されやすい。
- 日本語の文字を使っているので、小学生からでも抵抗なく扱える。また、全角/半角の区分をしないよう配慮しているので、文字種別に起因するエラーに遭遇しにくい。
- オブジェクト指向に基づいていて、さまざまなオブジェクトを使うことで普段接しているプログラムに近いことができる。また、オブジェクトの概念を学べる。

このほか、この資料では扱っていませんが、ドリトルの開発者の人たちはロボットカーや Arudino などの汎用制御基盤を用いた実践も多くおこなっています。計測・制御に関わる内容を扱いたい場合は、そのようなものもぜひ調べてみてください。

## 3.7 open quetions

- テキスト型の言語についてどう思いますか? ビスケットなどのビジュアル型とくらべてハードルが高いと思いましたか?
- ドリトルのような「書き方の規則」を守るのに費す労力はどれくらい大きいでしょうか? 構文エラーを直すのには手間が掛かりますか? または慣れれば問題ないでしょうか?
- 日本語を用いた言語では「日本語とプログラムの区別がはっきりしない」という弱点が指摘されることがありますが、この点はどうだったでしょうか?
- 「既存の名前」を指定することで色々な機能を使える反面、どのような「既存の名前」があるか分からないで不安になるという弱点もあります。このトレードオフをどう思いますか。
- 連接・反復・分岐という制御構造の基本をこのような扱いで学べますか (今回の内容では制御構造よりもほかのことに力点を置いています)。
- 絵を見せて、その絵と同じものを作らせる、という形の演習は有効でしょうか?
- 変数の導入として「絵の辺の長さの制御」を題材にしていますが、これについてはどうでしょうか?
- 高速とか正確とか何回やっても同じ結果などのコンピュータの性質はこの内容で充分学べるでしょうか?
- GUI 部品などの「普段目にするもの」が扱えることはどれくらい有効でしょうか?
- イベントトリブンの考え方はすんなり受け入れられるでしょうか?
- ドリトルの制御構造の書き方は普通の言語とかなり違いますが、そのあたりは問題ないでしょうか? 制御構造を学ぶ際の難しさについてはどうでしょうか?
- アニメーションの原理や対話的アニメーションの仕組みなどは理解してもらえそうでしょうか?
- 人間の特性に対する理解や、ゲームの難しさとの関係についてはどうでしょうか?

## まとめ

本講座では、まず第1章でプログラミング学習の基本的枠組について、目的や望ましい進め方などを中心に指針をまとめ、続いてその具体例として第2章でビスケット、第3章でドリトルというそれぞれの特徴を持った教育向けプログラミング言語/環境を用いた授業案を紹介しました。いずれも、授業案とその考え方に加えて、そこで (プログラミング学習の目的である) コンピュータとソフトウェアの原理理解について気づけることを多く取り上げました。

実際に教室でこれらの題材をもとに授業をする場合には、子どもたちがプログラミングを楽しいと思い、「自分の課題に熱中」することに、もっとも重点を置いて頂きたいと思います。

そのためには、サポートしてくれるスタッフの有無などにもよりますが、内容を詰め込まずにゆったりとした時間配分で実施することが必要です。本講座では時間の関係でかなり詰め込んだ授業案をおみせしていますが、実際にはもっと時間を掛けて実施するのが適切なものも多くあります。それぞれの環境に応じて調整してください。