プログラミングプロジェクト＃1 — 始めよう

久野　靖 (電気通信大学)

2021.7.11

今回は初回ですが、次のことが目標となります。

- プログラムとは何であるか理解し、簡単な Ruby のプログラムを動かせるようになる。
- 一本道の (分岐や反復のない) 手順を考えられるようになる。
- 簡単な (円と長方形から成る) 画像が作れるようになる。

1　プログラムとモデル化

1.1　モデル化とコンピュータ

モデル (model) とは、何らかの扱いたい対象があって、その対象全体をそのまま扱うのが難しい場合に、その特定の側面 (扱いたい側面) だけを取り出したものを言います。

たとえば、プラモデルであれば飛行機や自動車などの「大きさ」「重さ」「機能」などは捨てて「形」「色」だけを取り出したもの、と言えます。ファッションモデルであれば、さまざまな人が服を着る、その「様々さ」を捨てて特定の場面で服を見せる、という仕事だと言えます。

コンピュータで計算をするのに、なぜモデルの話をしているのでしょう? それは、コンピュータによる計算自体がある意味で「モデル」だからです。たとえば、「三角形の面積を求める」という計算を考えてみましょう。底辺が 10cm、高さが 8cm であれば

$$\frac{10 \times 8}{2} = 40(cm^2)$$

ですし、底辺が 6cm、高さが 5cm であれば

$$\frac{6 \times 5}{2} = 15(cm^2)$$

です。「電卓」で計算するのなら、実際にこれらを計算するようにキーを叩けばよいわけです。

1	0	×	8	÷	2	=

しかし、コンピュータでの計算はこれとは違っています。なぜなら、コンピュータは非常に高速に計算するためのものなので、いちいち人間が「計算ボタン」を押していたら人間の速度でしか計算が進まず意味がないからです。

そこで、「どういうふうに計算をするか」という**手順** (procedure) を予め用意しておき、実際に計算するときは**データ** (data) を与えてそれからその手順を実行させるとあっという間に計算ができる、というふうにします。そしてこの手順が**プログラム** (program) です。

たとえば面積の計算だったら、手順は

☆	×	◇	÷	2	=

みたいに書いてあり、あとで「☆は 10、◇は 8」というデータを与えて一気に計算します (もちろん、「☆は 6、◇は 5」とすれば別の三角形の計算ができます)。これを捉え直すと、「個々の三角形の面積の計算」から「具体的なデータ」を取り除いた「計算のモデル」が手順だ、ということになります。[1]

コンピュータでの計算はモデル、と言うのには別の意味もあります。三角形は 3 つの直線 (正確に言えば線分) から成りますが、世の中には完璧な直線など存在しませんし、まして鉛筆で紙の上に引いた線は明らかに「幅」を持っていて縁はギザギザ曲がっています。また、10cm とか 8cm とか「きっかり」の長さも世の中には存在しません。でも、そういう細かいことは捨てて「理想的な三角形」に抽象化してその面積を考えて計算しているわけです。

逆に言えば、コンピュータでの計算は常に、現実世界をそのまま扱うのではなく、必要な部分をモデルとして取り出し、それを計算している、ということです。この意味での抽象化やモデル化には、皆様はこれまで数学を通して多く接してきたと思いますが、これからはコンピュータでプログラムを扱う時にもこのようなモデル化を多く扱っていきます。

1.2 アルゴリズムとその記述方法

「三角形の面積の計算方法」のような、計算 (や情報の加工) の手順のことを**アルゴリズム** (algorithm) と言います。ある手順がアルゴリズムであるためには、次の条件を満たす必要があります。

- 有限の記述でできている。
- 手順の各段階に曖昧さがない。
- 手順を実行すると常に停止して求める答えを出す。[2]

1 番目は、「無限に長い」記述は書くこともコンピュータに読み込ませることも不可能だからです。2 番目は、曖昧さがあるとそれをコンピュータで実行させられないからです。3 番目はどうでしょうか。実際にコンピュータのプログラムを書いてみると、手順に問題があって実行が止まらなくなることも頻繁に経験しますが、そのようなものはアルゴリズムとは言えないのです。[3]

アルゴリズムを考えたり検討するためには、それを何らかの方法で記述する必要があります。その記述方法には色々なものがありますが、ここでは手順や枝分かれ等をステップに分けて日本語で記述する、**擬似コード** (pseudocode) と呼ばれる方法を使います。**コード** (code) とは「プログラムの断片」という意味で、「擬似」というのはプログラミング言語ではなく日本語を使うから、ということです。

三角形の面積計算のアルゴリズムを擬似コードで書いてみます。[4]

- triarea: 底辺 w、高さ h の三角形の面積を返す
- s $\leftarrow \frac{w \times h}{2}$。
- 面積 s を返す

1.3 変数と代入/手続き型計算モデル

上のアルゴリズム中で次のところをもう少しよく考えてみましょう。

- s $\leftarrow \frac{w \times h}{2}$。

[1] モデルを作る時の「不要な側面を捨てる」という作業を**抽象化** (abstraction) と言います。つまり、具体的な計算を抽象化したものが手順、という言い方をしてもよいわけです。

[2] 実は、計算の理論の中に「答えを出すかどうか分からないが、出したときはその答えが正しい」という手順を扱う部分もありますが、ここでは扱いません。

[3] 停止することを条件にしておかないと、アルゴリズムの正しさについて論じることが難しくなります。たとえば、「このプログラムは永遠に計算を続けるかもしれませんが、停止したときは億万長者になる方法を出力してくれます」と言われて、それを実行していつまでも止まらない (ように思える) とき、上の記述が正しいかどうか確かめようがありません。

[4] 以下ではこのように、何を受け取って何を行う手順 (アルゴリズム) かを明示するようにします。上の例で「返す」というのは、底辺と高さを渡されて計算を開始し、求まった結果 (面積) を渡されたところに答えとして引き渡す、というふうに考えてください。

この「←」は**代入** (assignment) を表します。代入とは、右辺の**式** (expression) [5] で表された値を計算し、その結果を左辺に書かれている**変数** (variable — コンピュータ内部の記憶場所を表すもの) に「格納する」「しまう」ことを言います。つまり、「w と h を掛けて、2 で割って、結果を s のところに書き込む」という「動作」を表していて、数式のような定性的な記述とは別物なのです。

数式であれば $s = \frac{w \times h}{2}$ ならば $h = \frac{2s}{w}$ のように変形できるわけですが、アルゴリズムの場合は式は「この順番で計算する」というだけの意味、代入は「結果をここに書き込む」というだけの意味ですから、そのような変形はできないので注意してください (困ったことに、多くのプログラミング言語では代入を表すのに文字「=」を使うので、普通の数式であるかのような混乱を招きやすいのです)。

モデルという立場からとらえると、式は「コンピュータ内の演算回路による演算」を抽象化したもの、変数は「コンピュータ内部の**主記憶**ないし**メモリ** (memory) 上のデータ格納場所」を抽象化したもの、そして代入は「格納場所へのデータの格納動作」を抽象化したもの、と考えることができます。

このような、式による演算とその結果の変数への代入によって計算が進んでいく計算のモデルを**手続き型計算モデル**と呼び、そのようなモデルに基づくプログラミング言語を**命令型言語** (imperative language) ないし**手続き型言語** (procedural language) と呼びます。手続き型計算モデルは、今日のコンピュータとその動作をそのまま素直に抽象化したものになっています。このため手続き型計算モデルは、最も古くからある計算モデルでもあるのです。

コンピュータによる計算を表すモデルとしては他に、関数とその評価に土台を置く関数型モデルや、論理に土台を置く論理型モデルなどもあるのですが、上記のような理由から、手続き型モデルが今のところもっとも広く使われています。

2 アルゴリズムとプログラミング言語

2.1 プログラミング言語

プログラムとは、アルゴリズムを実際にコンピュータに与えられる形で表現したものであり、その具体的な「書き表し方」ないし「規則」のことを**プログラミング言語** (programming language) と呼びます。これはちょうど、人間が会話をする時の「話し方」として日本語、英語など多くの言語があるのと同様です。ただし、**自然言語** (natural language — 日本語や英語など、人間どうしが会話したり文章を書くのに使う言語) とは違い、プログラミング言語はあくまでもコンピュータに読み込ませて処理するための人工言語であり、書き方も杓子定規です。

ひとくちにプログラミング言語といっても、実際にはさまざまな特徴を持つ多くのものが使われています。ここでは、プログラムが簡潔に書けて簡単に試して見られるという特徴を持つ、**Ruby** という言語を用いてゆきます。

2.2 Ruby 言語による記述

では、三角形の面積計算アルゴリズムを Ruby プログラムに直してみましょう。本クラスでは入力と出力は基本的に **irb コマンド** [6] の機能を使わせてもらって楽をするので、計算部分だけを Ruby の**メソッド** (method) [7] として書くことにします。先にアルゴリズムを示した、三角形の面積計算を行うメソッドは次のようになります。

```
def triarea(w, h)
  s = (w * h) / 2.0
  return s
end
```

詳細を説明しましょう。

[5] プログラミングで言う式とは、計算のしかたを数式に似た形で記述したものを言います。先に説明した、電卓で計算する手順を記したようなものと思ってください。

[6] Ruby の実行系に備わっているコマンドの 1 つで、さまざまな値をキーボードから入力し、それを用いてプログラムを動かす機能を提供してくれます。

[7] メソッドは他の言語での**手続き・サブルーチン** (subroutine) に相当し、一連の処理に名前をつけたもののことです。

1. 「def メソッド名」〜「end」の範囲が 1 つのメソッド定義になる。
2. メソッド名の後に丸かっこで囲んだ名前の並びがあれば、それらは**パラメタ** (parameter) ないし**引数**[8] の名前となる。メソッドを呼び出す時、各パラメタに対応する値を指定する。
3. メソッド内には**文** (statement — プログラムの中の個々の命令のこと) を任意個並べられる。各々の文は行を分けて書くが、1 行に書く場合は「;」で区切る。たとえば上の例のメソッド本体を 1 行に書きたければ「s = (w * h) / 2.0; return s」とする。
4. 文は原則として先頭から順に 1 つずつ実行される。
5. **return 文**「return 式」を実行するとメソッド実行は終了 (式の値がメソッドの結果となる)。

上の例は擬似コードに会わせるように、面積の計算結果を変数 s に入れてからそれを return していましたが、return の後ろに計算式を直接書くこともできるので、次のようにしても同じです。

```
def triarea(w, h)
  return (w * h) / 2.0
end
```

このように、たったこれだけのコードでも、大変細かい規則に従って書き方が決まっていることが分かります。要は、プログラミング言語というのはコンピュータに対して実際にアルゴリズムを実行する際のありとあらゆる細かい所まで指示できるように決めた形式なのです。

そのため、プログラムのどこか少しでも変更すると、コンピュータの動作もそれに相応して変わるか、(もっとよくある場合として) そういうふうには変えられないよ、と怒られることになります。いくら怒られても偉いのは人間であってコンピュータではないので、そういうものだと思って許してやってください。

2.3 動かしてみよう!

ではプログラムを動かしてみます。まず、テキストエディタ (メモ帳など) で上と同じ内容を sample1.rb というファイルに打ち込んで保存してください。この、人間が打ち込んだプログラムを (プログラムを動かす「源」という意味で) **ソース**ないし**ソースコード** (source code) と呼びます。Ruby のソースファイルは最後を「.rb」にするのが通例です。そして、先に進む前に、作成したファイルがその場所にあることを確認してください。ディレクトリが違っている場合はソースファイルのあるディレクトリに移動しておくこと。

いよいよ、このコードを動かしてみましょう。前述のように、この講義では **irb** というコマンドを通じて Ruby プログラムをファイルから読み込んだり実行解しさせます。irb の動かし方はシステムによって違うので担当教員に教わってください。以下の例ではまず irb を起動するところから示しています (「%」はプロンプト文字列のつもりなので打ち込まないでください)。

```
% irb
irb(main):001:0>
```

この「irb なんとか>」というのは irb の**プロンプト** (prompt — 入力をどうぞ、という意味の表示) で、ここの状態で Ruby のコードを打ち込めます。

プロンプトの読み方を説明すると、main というのは現在打ち込んでいる状態がメインプログラム (最初に実行される部分) に相当することを意味しています。次の数字は何行目の入力かを表しています。最後の数字はプログラムの**入れ子** (nesting — 「はじめ」と「おわり」で囲む構造の部分) の中に入るごとに 1 ずつ増え、出ると 1 ずつ減ります。とりあえずあまり気にしなくてよいでしょう。以後の実行例では見た目がごちゃごちゃしないように「irb>」だけを示すことにします。

[8] メソッドを使用するごとに、毎回異なる値を引き渡して、それに基づいて処理を行わせるための仕組みです。

次に load(ファイルからプログラムを読み込んでくる、という意味です) で sample1.rb を読み込ませます。ファイル名は**文字列** (string) として渡すので'' または""で囲んでください。[9]

```
irb> load 'sample1.rb'
=> true
irb>
```

true が表示されたら読み込みは成功で、ファイルに書かれているメソッド triarea が使える状態になります。成功しなかった場合は、ファイルの置き場所やファイル名の間違い、ファイル内容の打ち間違いが原因と思われるので、よく調べて再度 load をやり直してください。

なぜわざわざ 3〜4 行程度の内容を別のファイルに入れて面倒なことをしているのでしょうか? それは、メソッド定義の中に間違いがあった時、定義を毎回 irb に向かって打ち直すのでは大変すぎるからです。このため、以下でもメソッド定義はファイルに入れて必要に応じて直し、irb では load とメソッドを呼び出して実行させるところだけを行う、という分担にします。

load が成功したら triarea が使えるはずなので、それを実行します。

```
irb> triarea 8, 5
=> 20.0
irb> triarea 7, 3
=> 10.5
irb>
```

確かに実行できているようです。irb は quit で終わらせられます。

```
irb> quit
%
```

苦労の割に大した結果ではない感じですが、初心者の第 1 歩として、着実に進んでいきましょう。

演習 1 例題の三角形の面積計算メソッドをそのまま打ち込み、irb で実行させてみよ。数字でないものを与えたりするとどうなるかも試せ。

演習 2 三角形の面積計算で、割る数の指定を「2.0」でなくただの「2」にした場合に何か違いがあるか試せ。

演習 3 次のような計算をするメソッドを作って動かせ。[10]

a. 2 つの実数を与え、その和を返す (ついでに、差、商、積も)。何か気づいたことがあれば述べよ。

b. 「%」という演算子は**剰余** (reminder) を求める演算である。上と同様に剰余もやってみよ。何か気づいたことがあれば述べよ。

c. 円錐の底面の半径と高さを与え、体積を返す。[11]

d. 実数 x を与え、x の平方根を出力する。さまざまな値について計算し、小数点以下何桁くらい表示されるか (計算の精度がどれくらいあるか) 検討せよ。[12]

e. その他、自分が面白いと思う計算を行うメソッドを作って動かせ。

[9] 本来ならメソッドに渡すパラメタは丸かっこで囲むのですが、Ruby では曖昧さが生じない範囲でパラメタを囲む丸かっこを省略できます。本資料ではプログラム例の丸かっこは省略しませんが、irb コマンドに打ち込む時は見た目がすっきりするので丸かっこを適宜省略します。

[10] 1 つのファイルにメソッド定義 (def ... end) はいくつ入れても構わないので、ファイルが長くなりすぎない範囲でまとめて入れておいた方が扱いやすいと思います。

[11] 円錐の体積は「底面積×高さ× $\frac{1}{3}$」。底面積は「$2\pi r^2$」。

[12] x の**平方根** (square root) は **Math.sqrt**(x) で計算できます。または x の 0.5 乗 (x ** 0.5) としても計算できます。

3　画像を生成する

ここまで、プログラムの出力は数値ばかりでした。コンピュータが最初に作られた目的は数値の計算のためでしたからそれは自然なのですが、やはり数値だけではつまりません。そこで最後に、簡単な絵を生成するという例題を見てみます。ただし、まだ Ruby の学習に入ったばかりですので、絵を生成するために利用する下請けのメソッド等は説明しません。最後の (絵を生成する) メソッド mypict1 だけ読んでください。

```
$img = Array.new(200) do Array.new(300) do [255,255,255] end end
def pset(x, y, r, g, b)
  if 0 <= x && x < 300 && 0 <= y && y < 200
    $img[y][x][0] = r; $img[y][x][1] = g; $img[y][x][2] = b
  end
end
def writeimage(name)
  open(name, "wb") do |f|
    f.puts("P6"); f.puts("300 200"); f.puts("255")
    $img.each do|a| a.each do |p| f.write(p.pack("ccc")) end end
  end
end
def fillrect(x0, y0, w, h, r, g, b)
  (y0-h/2).to_i.step(y0+h/2) do |y|
    (x0-w/2).to_i.step(x0+w/2) do |x| pset(x, y, r, g, b) end
  end
end
def fillcircle(x0, y0, rad, r, g, b)
  (y0-rad).to_i.step(y0+rad) do |y|
    (x0-rad).to_i.step(x0+rad) do |x|
      if (x-x0)**2 + (y-y0)**2 <= rad**2 then pset(x, y, r, g, b) end
    end
  end
end

def mypict1
  fillcircle(60, 90, 30, 255, 0, 0)
  fillrect(100, 60, 80, 60, 0, 255, 0)
  writeimage("pict1.ppm")
end
```

ここで使っている機能は以下の 3 つです。色の指定には赤 (r)、緑 (g)、青 (b) の強さを 0〜255 の整数で指定します。

- fillcircle(x, y, rad, r, g, b) — 中心 (x, y)、半径 rad の円を rgb で塗る。
- fillrect(x, y, w, h, r, g, b) — 中心 (x, y)、幅 w、高さ h の長方形を rgb で塗る。
- writeimage(ファイル名) — 画像をファイルに出力。ファイル名は「.ppm」で終わる文字列。

これを load 後 mypict1 を実行します。それで mypict を読むと、まず $(60, 90)$ に半径 30 の真っ赤な円を描き ((255,0,0) 赤のみ最大なので真っ赤)、その上に中心 $(100, 60)$、幅 80、高さ 60 の緑の長方形 ((0,255,0) は真緑) を重ね、最後にファイル mpict1.ppm に出力しています (図 1)。Y 軸の向きと、後から描いたものが「上」に重なることに注意。

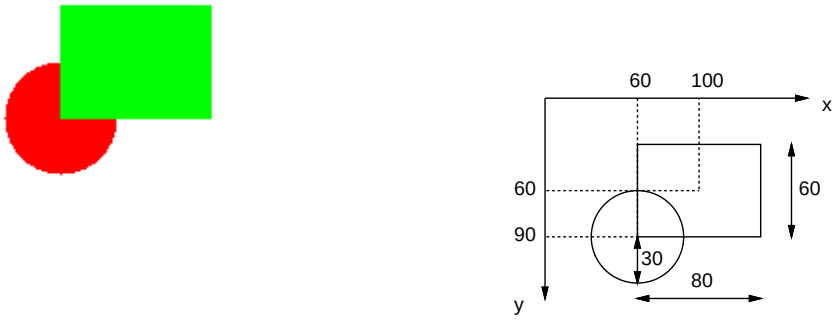
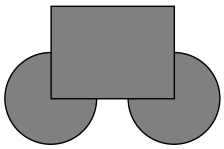
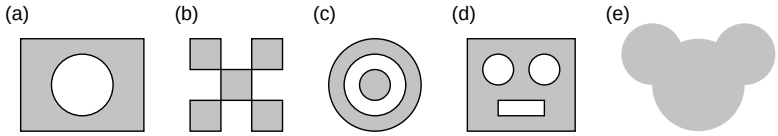


図 1: 最初の画像

演習 4 例題プログラムの `mypict1` をそのまま動かし、生成される画像を確認しなさい (画像の表示方法は環境により異るが、「`display pict1.ppm`」「`gimp pict1.ppm`」などでできる環境が多いはず)。確認できたら、タイヤ (円) をもう 1 つ増やして車の絵にしてみなさい。色は変えてみるとよい。



演習 5 以下のような図形を描くプログラムを作成してみなさい (色は自由に決めてよいです)。



演習 6 長方形と円を組み合わせた自分の好きな絵を考え、それを描くプログラムを作ってみなさい。

本日の課題 1A

「演習 3」「演習 4」で動かしたプログラム (どれか 1 つでよい) を含むレポートを提出しなさい。プログラムと、簡単な説明が含まれること。アンケートの回答もおこなうこと。

Q1. プログラム、って恐そうですか? 第 2 外国語と比べてどう?

Q2. Ruby 言語のプログラムを打ち込んで実行してみて、どのような感想を持ちましたか?

Q3. 本日の全体的な感想と今後の要望をお書きください。

次回までの課題 1B

「演習 4」、「演習 5」(の小課題)、「演習 6] から 2 つ選択してプログラムを作り (ただし 1A で提出したものは除外、以後も同様)、レポートを提出しなさい。プログラムと、課題に対する報告・考察 (やってみた結果・そこから分かったことの記述) が含まれること。アンケートの回答もおこなうこと。

Q1. プログラムを作るという課題はどれくらい大変でしたか?

Q2. 実際に複数のプログラムを打ち込んで動かしてみて、想像していたのと違うところはありましたか。

Q3. 課題に対する感想と今後の要望をお書きください。