プログラミングプロジェクト＃4 – 構造化グラフィクス、配列

久野　靖 (電気通信大学)

2018.8.27

1　はじめに

今回の内容としては、次のものを取り上げます。

- 手続きを用いた絵の描画の構造化
- データ構造と配列

2　前回演習問題の解説

2.1　演習 1a、演習 1b — while 文

演習 1a は、絵を斜めに動かせ、というものでした。それは簡単で、x だけでなく y も変化させればいいわけです。コードを示します (ライブラリは省略)。

```
def car(u, x, y, r1, g1, b1, r2, g2, b2)
  fillcircle(x-3*u, y+2*u, 2*u, r2, g2, b2)
  fillcircle(x+3*u, y+2*u, 2*u, r2, g2, b2)
  fillrect(x, y, 6*u, 4*u, r1, g1, b1)
end
def mypicture41
  x = 30; y = 80
  while x <= 300 do
    car(8, x, y, 120, 40, 180, 250, 200, 20)
    x = x + 80; y = y + 20
  end
  writeimage('pict41.ppm')
end
```

絵は図 1 の左に掲載します。

演習 1b は、2 つの車を動かせ、ということでした。そこで、while ループを 2 つにして、それぞれ先と同様に動かしています。while ループ 1 つに 2 つの car 呼び出しでできると思うかも知れません。それでも確かにできるのですが、描ける車の数が 2 台とも同じになってしまいます。

```
def mypicture42
  x = 30; y = 80
  while x <= 300 do
    car(8, x, y, 120, 40, 180, 250, 200, 20)
    x = x + 80; y = y + 20
  end
  x2 = 20; y2 = 160
  while x2 <= 300 do
```

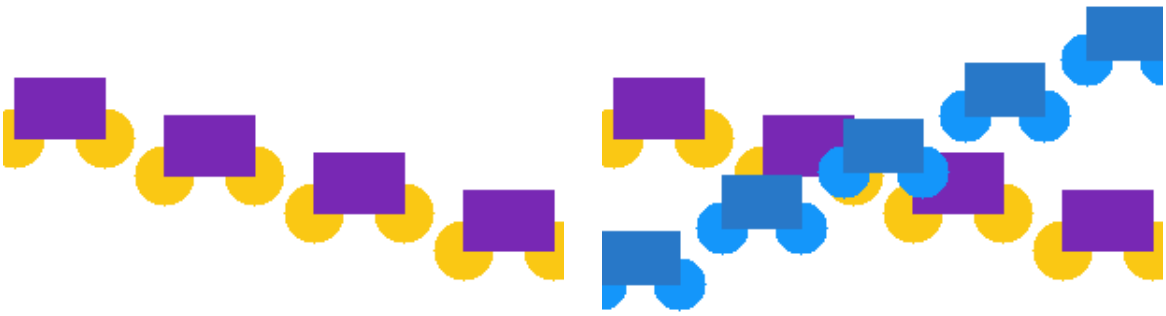

図 1: while 文の演習

```
    car(7, x2, y2, 40, 120, 200, 20, 150, 250)
    x2 = x2 + 65; y2 = y2 - 30
  end
  writeimage('pict42.ppm')
end
```

絵は図 1 の右に掲載します。確かに青い車が 1 台多く描けていますね。

2.2 演習 2 — 計数ループ

これはいずれも、赤い正方形を並べて作りました。最後の 2 つはやや難しかったと思います。

```
def mypicture42a
  5.times do |i|
    fillrect(20*(i+5), 100, 10, 10, 255, 0, 0)
  end
  writeimage("pict.ppm")
end
def mypicture42b
  5.times do |i|
    fillrect(20*(i+5), 20*(i+3), 20, 20, 255, 0, 0)
  end
  writeimage("pict.ppm")
end
def mypicture42c
  5.times do |i|
    fillrect(20*(i+5), 100, 10, 60, 255, 0, 0)
  end
  writeimage("pict.ppm")
end
def mypicture42d
  3.times do |i|
    fillrect(20*(i+5), 40, 10, 10, 255, 0, 0)
    fillrect(140, 20*(i+3), 10, 10, 255, 0, 0)
    fillrect(20*(-i+6), 100, 10, 10, 255, 0, 0)
    fillrect(80, 20*(-i+4), 10, 10, 255, 0, 0)
  end
```

```
  writeimage("pict.ppm")
end
def mypicture42e
  7.times do |i|
    fillrect(20*(i+5), 20*(i+2), 20, 20, 255, 0, 0)
    fillrect(20*(i+5), 20*(-i+8), 20, 20, 255, 0, 0)
  end
  writeimage("pict.ppm")
end
```

3 構造化グラフィクス

そろそろ、込み入った絵を描いてもいいころです。そこで手続きを用いた込み入った絵の描画の考え方をまとめておきます。その要点は次のようなことです。

- メインとなる手続きは、主要な部品となる手続きを呼び出し、出来上がった絵を書き出す。
- 主要な部品は、幾つかの基本的な絵を組み合わせて出来ている。場合によっては、複雑な主要部品は、下請けの部品を組み合わせて作られているかも知れない。
- 下請けの部品 (あれば) は、そのまた下請けの部品や、基本図形を組み合わせて出来ている。
- 最後に、基本図形や書き出しの機能は、ライブラリとして用意されている。

言い替えれば、このようなやり方で、絵の全体から細かい部分へ、考える部分を分けるのと同時に、手続きも分割して行くのです。これを「構造化グラフィクス」と呼び、絵のプログラムを見通しよくする効果があります。

説明だけだと分かりにくいでしょうから、サンプルを示しましょう (図 2)。この絵は「海をゆく船」をあらわすもので、「船」「波」「空」が主要な部品となっています。

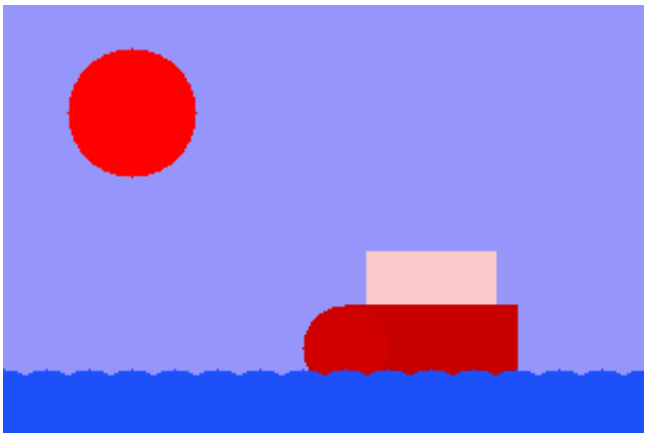

図 2: 海を行く船

図 3 にあるように、絵の本体は shiponthesea というメソッドですが、これは 3 つの下請けメソッドを呼び出して、できた絵を出力するだけです。

```
def shiponthesea
  sky(150, 150, 250, 255, 0, 0); ship(200, 160, 10); wave
  writeimage('pict4.ppm')
end
```

下請けメソッドは、空、船、波の順で呼び出します (手前に来るものが後)。そして最後に、絵を書き出します。空と太陽の色、船の位置と大きさは指定するようになっています。

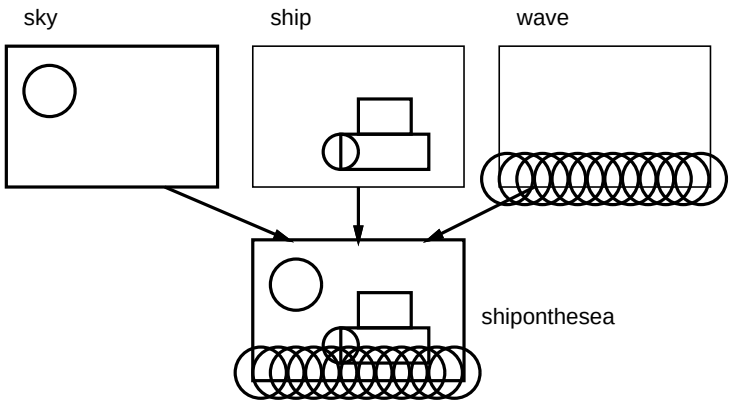

図 3: 絵の分解

メソッド sky は、背景となる空を指定された色で塗り、それから太陽を赤い円で描きます。空の色だけ変えられるようにしたのは、場面によって色を変えたいだろうと思ったからです。

```
def sky(r1, g1, b1, r2, g2, b2)
  fillrect(150, 100, 300, 200, r1, g1, b1)
  fillcircle(60, 50, 30, r2, g2, b2)
end
```

メソッド ship は、円と長方形を 2 つ組み合わせて船の形を描きます。船は位置 x, y と大きさ u を指定します。

```
def ship
  fillrect(x, y-3*u, 6*u, 3*u, 250, 200, 200)
  fillrect(x, y, 8*u, 4*u, 200, 0, 0)
  fillcircle(x-4*u, y, 2*u, 2000, 0, 0)
end
```

最後にメソッド wave は、ループを使って円をたくさん描くことで波を表しています。これは指定するものはありません。

```
def wave
  17.times do |i|
    fillcircle(i*20, 190, 20, 30, 80, 250)
  end
end
```

これくらいだと、メソッドを分けなくても一辺に描いてしまえそうですが、あとで絵の内容を増やしたり、手直しする時には、構造化グラフィクスの考え方に従った方が結局やりやすいのです。以下にライブラリを含めた全体を示しておきます。

```
$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).step(y0+h/2) do |y|
    (x0-w/2).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).step(y0+rad) do |y|
    (x0-rad).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 sky(r1, g1, b1, r2, g2, b2)
  fillrect(150, 100, 300, 200, r1, g1, b1)
  fillcircle(60, 50, 30, r2, g2, b2)
end
def ship(x, y, u)
  fillrect(x, y-3*u, 6*u, 3*u, 250, 200, 200)
  fillrect(x, y, 8*u, 4*u, 200, 0, 0)
  fillcircle(x-4*u, y, 2*u, 2000, 0, 0)
end
def wave
  17.times do |i|
    fillcircle(i*20, 190, 20, 30, 80, 250)
  end
end
def shiponthesea
  sky(150, 150, 250, 255, 0, 0); ship(200, 160, 10); wave
  writeimage('pict4.ppm')
end
```

演習 1 海を行く船の例題をそのまま動かしなさい。うまく行ったら、さらに次のように直してみなさい。

a. 太陽を月に変えて、夜景にする。
b. 船の位置をかえたり、船ふやす。
c. 波の色を変えられるようにする。また、1 つおき (または 2 つおき) に違った色にする。
d. 空に雲をたなびかせる。または船の煙突から煙をださせる。

4　配列とその利用

4.1　データ構造の概念と配列

ここまででではプログラムが扱うデータは個々の「値」であり、1 つの変数に 1 つの値が入っていました。しかしこのやり方では、大量のデータを扱うのが困難なのは明らかです。ではどうするかというと、複数のデータを組にしたり、列として並べるなどの「構造」を持たせて扱う、というのが答えです。この、データに持たせる構造のことを**データ構造** (data structures) と言います。

プログラミング言語の用語では、データの種類のことを**データ型** (data types)、その中で「整数」「実数」など単一の値から成るものを**基本データ型** (primitive data types) と呼びます。それと対比して、組や列など複数の値が集まったデータのことは**複合データ型** (compound data types) と呼びます。実は文字列は、中に複数の文字が含まれているので複合データ型だといえます。

今回は複合データ型のうちでもよく使われる**配列** (array) を取り上げます。配列は既に「`[1, 2, 3]` のように値を並べたもの」として言及したことがありますが、要するに値が一列に並んだものです。図 4 のように、整数であれば 1 つの変数に 1 つの値しか入れられませんが、配列を使うことで 1 つの変数に一連の値を入れることができます。

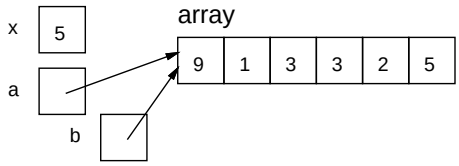

図 4: 配列の概念

図 4 を見て不思議に思ったことはないでしょうか。具体的には、基本型では変数の位置に「箱」が書かれていてそこに値が入っていますが、複合型では少し離れたところにデータを入れる場所があって、変数からはそこに矢印が出ています。

実はこの矢印はデータのありかを示す**参照** (reference — ありかを指す値で、実体はメモリ上の番地だと思ってよい) です。そして、変数に配列を入れると、配列本体はどこか別の場所に置かれ、変数にはその場所への参照が入ります。そして、「`b = a`」のように変数間で代入をした時、基本型では値 (箱の中身) がコピーされますが、複合型では参照 (矢印) がコピーされるだけで、本体は 1 つのまま (単に 2 つの変数が同じ場所を指すだけ) です。[1]

さらに、「2 つの変数が同じ場所を指している」状態でその複合データの中身を書き換えると、複合データは 1 つだけなので、どちらの変数から見た複合データも同じように変化していることになります。このあたりの挙動は間違えやすいので注意が必要です。

4.2　配列の生成

配列を使うには、まず配列を作り出す必要があります。その方法が色々ありますので、ここではそれらについて説明しておきます。

```
a = [1, 2, 3]           # 直接指定
a = Array.new(100, 0)   # 要素数と初期値
a = Array.new(100) do 0 end  # 要素数とブロック
a = Array.new(100) do |i| 2*i end  # 〃
```

1 番目の方法はこれまでにも使ってきた、各要素を直接指定する方法です。[2] この方法は、比較的少数の値を用意する場合に使います。

[1] Ruby の場合。C 言語はまた違います。

[2] 値を並べて書く方法は「そのまま値を書く」ことから「配列リテラル」と呼ぶこともあります。しかし、配列では初期値を指定するのに変数や任意の式を指定できるので、厳密に言えば「そのまま」ではありません。Ruby の用語でもこの書き方は**配列式** (array expression) というのが正式な呼び方です。

2 番目は、要素数と初期値を指定する方法で、要素数の多い配列を用意するときにはこの方法が一番単純です。[3]

3・4 番目も、要素数と初期値を指定する方法ですが、初期値として値を計算するブロック (do〜end) を指定するところが違います (この場合はブロックの中で式を直接指定します。メソッドではないので return は書けません)。0 などと定数を指定した場合は 2 番目と変わりませんが、ブロックは (times などと同様)「何番目」というパラメタを受け取ることができるので、それを用いて計算により初期値を決めてもよいのです。

配列は後からメソッド `push` で要素を追加できます。上の例の 4 番目と次は同じ結果になります。

```
a = []                          # 0 要素の配列を作り
100.times do |i| a.push(2*i) end  # 0〜198 を追加
```

現在の**配列の長さ** (要素数) は、メソッド `length` で取得できます。上の例では `a.length` は 100 です。

演習 0 ブロックを指定する形で 10 要素の配列を生成し、初期値を (a)〜(d) のようにしなさい。

(a)

10	9	8	7	6	5	4	3	2	1

(b)

0	1	0	1	0	1	0	1	0	1

(c)

4	3	2	1	0	1	2	3	4	5

(d)

1	1	1	1	1	0	0	0	0	0

なお、初期値を指定するブロックの中でも if を使えます。

```
if 条件 then 式1 else 式2 end
```

この if は「if 式」であり、条件の成否に応じて「式 1」「式 2」のいずれかが全体の値となります。

4.3 配列の利用

いちど用意してしまえば、配列の個々の要素は 1 つの変数と同様に扱えます。ここで「どの要素か」を指定するのに `[...]` の中に式を書いて指定します。これを**添字** (index) と呼びます。たとえば上の例だと `a[0]`〜`a[99]` という要素があることになります (0 番目から数えることは慣れないと忘れやすいので注意してください)。

また、Ruby ではまだ用意していない添字番号 (たとえば上で「100 番」とか) の要素を参照すると `nil` が返ります。飛び離れた添字番号 (たとえば上で「200 番」とか) に値を格納すると、そこまでの途中の要素は全部 `nil` で埋められます。

では、配列を与えてその合計を求めるというのをやってみましょう (合計は積分とかで散々やったので簡単ですね)。

- arraysum：配列 a の数値の合計を求める
- 　sum ← 0。
- 　i を 0 から配列要素数の手前まで変えながら繰り返し、
- 　　sum ← sum + a[i]。
- 　繰り返し終わり。
- 　sum を返す。

Ruby コードは次のとおり。

[3] 初期値を指定しないと各要素の初期値は `nil` になります。

```
def arraysum(a)
  sum = 0
  a.length.times do |i|
    sum = sum + a[i]
  end
  return sum
end
```

一応、動かすところの様子を示します。

```
irb> arraysum([1,2,3,4,5])
=> 15
```

実は Ruby では「配列の各要素を取りながら周回するループ」というのもあって、そのほうが少し簡単になります。コードだけ示しておきます。

```
def arraysum1(a)
  sum = 0
  a.each do |x|      # x に配列の各要素が順次入る
    sum = sum + x
  end
  return sum
end
```

合計ならこのほうが少し簡単ですが、「何番目」を必要とする場合もあるので、その場合には計数ループを使うことになるでしょう。[4]

演習 2 上記の配列合計プログラムの好きな方をそのまま打ち込んで動かせ。動いたらこれを参考に下記のような Ruby プログラムを作れ。[5]

a. 数の配列を受け取り、その平均値を返す。
b. 数の配列を受け取り、その最大値を返す。
c. 数の配列を受け取り、最大値が何番目かを返す。なお先頭を 0 番目とし、最大値が複数あればその最初の番号が答えであるとする。
d. 数の配列を受け取り、最大値が何番目かを出力する。なお先頭を 0 番目とし、最大値が複数あればそれらをすべて出力する。
e. 数の配列を受け取り、その平均より小さい要素を出力する (例: 1、4、5、11 → 1、4、5)。
f. 数の配列を受け取り、その内容を「小さい順」に並べて出力する (例: 4, 11, 5, 1 → 1, 4, 5, 11)。

4.4 例題: 複数のボール

前回の演習 2 の中に、ボール (または別の図形) を四角く並べる、というのがありました。その手直し版を図 5 に載せます。素朴なやり方では、上、右、下、左と 4 つの辺をループで描くことになるでしょう。

しかし、これらを 4 つのボールが上、右、下、左辺に沿って動くと考え、各ボールの座標 X、Y と、1 きざみ当たりの動き DX、DY を、それぞれ 4 要素の配列に入れるようにすると、ずっと短いプログラムにできます。次のコードを見てください (ライブラリは省略)。

[4] メソッド `each_index` で配列の添字を順次取り出してループすることもできます。
[5] 「返す」の場合は上の例と同様に `return` を使い、「出力する」の場合は `puts` を使って画面に直接 (その場で) 出力させてください。`return` は使った瞬間にそのメソッド呼び出しは終ってしまうので、複数回 `return` を使うことはできません。

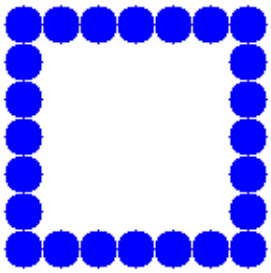

図 5: 4 つのボール

```
def fourballs
  x = [40, 160, 160, 40]; y = [40, 40, 160, 160]
  dx = [20, 0, -20, 0]; dy = [0, 20, 0, -20]
  x.length.times do |i|
    6.times do
      fillcircle(x[i], y[i], 10, 0, 0, 255)
      x[i] = x[i] + dx[i]; y[i] = y[i] + dy[i]
    end
  end
  writeimage('pict5.ppm')
end
```

どういう計算をしているか分かるでしょうか。最初のボールは `x[0],y[0]` ですから、最初の位置は (40,40) そして動きは `dx[0],dy[0]=(20,0)` ですから、引き続く座標は (60,40) (80,40) (100,40) (120,40) (140,40) (160,40) となります (最後の座標は計算するだけで表示なし)。残りのボールについても同様です。

演習 3 「複数のボール」の例題をそのまま動かして確認しなさい。`x[1],y[1]`、`x[2],y[2]`、`x[3],y[3]` の各ボールの座標も計算してみて確認すること。納得したら、次のことをやってみなさい。

a. 正方形をひとまわり小さく、また大きくしてみなさい。

b. ボールをもう 1 つ増やして「日」の字を描いてみなさい。

c. 今はどのボールも「6 回」描くようになっているが、もう一つ配列 `count[i]` を増やしてボールごとに繰り返しの回数が変えられるようにしなさい。その上で細長い四角を描かせなさい。

演習 4 構造化グラフィクスによる好きな絵を描きなさい。絵の一部に、配列を使った繰り返しを活用できるとなおよい。

本日の課題 4A

「演習 1」「演習 2」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. 制御構造の組み合わせができるようになりましたか。

Q2. 配列について学びましたが、使えそうですか。

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 4B

「演習 1」「演習 2」の (小) 課題から選択して 2 つ以上プログラムを作り、レポートを提出しなさい。配列の内容を含むことを強く勧めます。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. 配列が使いこなせるようになりましたか。

Q2. 配列を使うと「複数のもの」を統一的に扱えることに納得しましたか。

Q3. 課題に対する感想と今後の要望をお書きください。