情報科学 2007　久野クラス　＃4

久野　靖*

2007.11.2

はじめに

今回の主な内容は次の通りです。

- 手続き (メソッド) による抽象化と再帰関数
- レコード型、画像の生成

手続きが使えると自分のプログラムを見通しよく書けるようになりますので、意識して活用してください。そして、画像が生成できるとこれまでの数字ばっかりとは違ったものが扱えて楽しめると思いますので、それを次回までの課題にします。工夫して絵を作ってみてください。

1　演習問題解説

1.1　演習 2a — 区間 2 分法

区間 2 分法は自分でやってくれた人がほとんどだと思うけど一応。

- sqrt2bun：n の平方根を誤差 e で求める
- 　a ← 0。b ← n。
- 　$|a-b| >$ e である間繰り返し、
- 　　c ← $\frac{a+b}{2}$。
- 　　もし $c^2 > n$ なら、
- 　　　b ← c。
- 　　そうでなければ、
- 　　　a ← c。
- 　　枝分かれ終わり。
- 　繰り返し終わり。
- 　a を返す。

Ruby では次の通り。

```
def sqrt2bun(n, e)
  a = 0.0
  b = n
  while (a-b).abs > e do
    c = 0.5 * (a+b)
    if c**2 > n then b = c else a = c end
  end
  return a
end
```

*筑波大学大学院経営システム科学専攻

1.2 演習 2b — ニュートン法

ニュートン法の方が理屈は面倒だけど計算は枝分かれが無くて済むので簡単である。

- sqrt2newton：n の平方根を誤差 e で求める
- 　r ← 0。r1 ← n。
- 　$|r1 - r| > e$ である間繰り返し、
- 　　r ← r1。
- 　　r1 ← $\frac{r}{2} + \frac{n}{2r}$。
- 　繰り返し終わり。
- 　r を返す。

これの Ruby 版は次のとおり。

```
def sqrtnewton(n, e)
  r = 0.0
  r1 = n
  while (r1-r).abs > e do
    r = r1
    r1 = 0.5*r + n/(2.0*r)
  end
  return r
end
```

1.3 演習 3 — 配列の演習

演習 3 はアルゴリズムは略して Ruby のコードだけ記そう。まず最大。

```
def arraymax(a)
  max = a[0]
  a.each do |x|
    if x > max then max = x end
  end
  return max
end
```

このような形で配列を使う場合は、ふつうは「とりあえず max に最初の値を入れておき、より大きい値が出て来たら入れ替える」方法になる。なお、each は配列の各要素を順に取って来るループ。

次は最大の値が何番目に出て来るかなので、普通の計数ループにする。また、「何番目か」も変数に記録し、最大を更新したときにに同時に更新する。

```
def arraymaxno(a)
  max = a[0]
  pos = 0
  a.length.times do |i|
    if a[i] > max then max = a[i]; pos = i end
  end
  return pos
end
```

ところで、配列の各添字を順に扱う計数ループには a.length.times を使えばいいのだが、配列のメソッド a.each_index を使ってもよい。

2　手続き/関数

2.1　手続きと抽象化

プログラミング言語上の概念**手続き** (procedure) とは、ひとまとまりの動作に名前をつけたもの。そのひとまとまりを**呼び出す**ことによって何箇所からでも利用できる。また、呼び出すときに**パラメタ** (parameter) を渡すことで、呼び出す箇所ごとに動作の内容をいくらか調節できる。そして、Ruby では初回から作っているメソッドが手続きに相当する。

たとえば前回、「整数 n が素数かどうか調べる」というメソッドをまず作り、「素数を列挙する」というメソッドではそれを「呼び出して」いたことをご記憶だろうか。例題を少し手直ししたものを再掲する。

```
def isprime(n)
  2.step(Math.sqrt(n)) do |i|
    if n % i == 0 then return false end
  end
  return true
end

def primes(n)
  2.step(n, 2) do |i|
    if isprime(i) then puts(i) end
  end
end
```

このように 2 つのメソッドに分けることは、何がいいのだろう? それはもちろん、2 番目のメソッドが「もし i が素数なら」とひとことで書けるようになることである。

もう 1 つ別の例として「n までの間で、2 つ違いの素数が現れるとき、それらを打ち出す」ことを考える。これも上のメソッドがあれば、次のように書ける。

```
def adjacentprimes(n)
  2.step(n) do |i|
   if isprime(i) && isprime(i+2) then puts "#{i} #{i+2}" end
  end
end
```

このように、「もし i が素数で、かつ、i+2 が素数 なら」とひとことで言える。つまり、中では複雑な計算が必要な手順であったとしても、まとめて名前をつけることによって必要なら何箇所からでも呼び出せ、コードも分かりやすくなる。たとえば区間 2 分法のコードも、解を求める関数 `f()` を別に定義しておいて「`if(f(c) > 0) ...`」と書いたらずっと読みやすいですね?

これを難しく言うと、手続きを使うことで**抽象化**して考えられる (つまり繰り返しとかごちゃごちゃしたものはさて置いて、ともかく「n が素数かどうか」を調べるものがあると思うことができる) のが利点と言える。

2.2　手続き/関数と副作用

ここで「関数」と「手続き」について整理しておく。

- 数学的には、「手続き」というものは出てこない。「関数」とは「入力空間 (定義域) から出力空間 (値域) への写像」として定義される。
- プログラミング的には、「手続き」とは「ひとまとまりの動作 (手順) に名前をつけたもの」。そのうち「値を返すもの」(Ruby の場合は返す型として `void` 以外のものが指定されている場合) を「関数」と呼ぶことがある。
- そして、特定の言語…C、C++などでは「手続き」のことをすべて「関数」と呼ぶ。Ruby などでもそう呼ぶことがある。

- また、Ruby を含むオブジェクト指向言語では、オブジェクトに対する操作を行う手続きを「メソッド」と呼ぶ。これを援用して、すべての手続きを「メソッド」と呼ぶこともある。

そういうわけで呼び方が沢山あってややこしいが、ここでは当面「関数」「手続き」「メソッド」を適当に混ぜてつかう (適当ですみません)。

数学の「関数」とプログラミング言語の「関数」の最大の違いは、数学の関数では入力 (パラメタ) が同じなら結果は常に同じだが、プログラミング言語の関数はそうとは限らない。これは、実行する手順の中で変数などを書き換えてプログラム全体の状態を変化させることができるため。そのような動作のことを**副作用**と呼ぶ。たとえば次の例を見てみよう。

```
$x = 0

def myfunc(n)
  $x = $x + 1
  return n + $x
end
```

上の例では myfunc() はパラメタ n に変数$x の値を足した値を返す関数だが、1 度呼ばれるたびに$x を増やしているので (副作用)、n の値が同じでも呼ばれるごとに返す結果は変わって来るわけだ。

Ruby では$で始まる変数は**グローバル変数** (**広域変数**) と言い、手続きの外でも存在し続けている。そして上の例のように、手続きの中でグローバル変数を参照したり書き換えることもできる。これは用途によっては便利だけれど下手に使うと副作用のため分かりにくい問題が起きることがあるので注意した方がよい。

なお、これまでに使って来た (とくに指定のない) 変数はすべて、手続きの中だけで使える**ローカル変数** (**局所変数**) である。ローカル変数はその手続きが実行している間だけ存在していて、よその手続きからは参照できない。

手続きが副作用を持つのは、グローバル変数に対する書き換えだけとは限らない。たとえば puts などによる出力、パラメタとして渡された配列の中身の書き換えなどはすべて副作用である。

2.3 演習 3 — 配列の演習 (続き)

次は「配列要素のうち最大値は何番目にあるかを、すべて打ち出す」演習。「何番目か」を全部出力するので、まず最初に最大を求め、続いて番号を打ち出す、というふうに 2 回ループが必要になることに注意。ループ 1 つでは「現在のところ最大が見つかっても、もっと先に本当の最大があるかも知れない」ので無理。

```
def arraymaxnoall(a)
  max = arraymax(a)
  a.length.times do |i|
    if a[i] == max then puts(i) end
  end
end
```

最大は既に上で作ったので、それを利用している。この方が見やすいでしょう?

配列の最後の問題は上のと似ているが平均を求めてそれ以下ということで。平均は…前回合計を作ったのだから、それをデータ個数で割ればよい。

```
def arraysmallall(a)
  avg = arraysum(a).to_f / a.length;
  a.each do |x|
    if x < avg then puts(x) end
  end
end
```

しかしこの to_f というのは? それは、arraysum が返す合計は配列の中身がすべて整数だった場合、整数として返ってくる。それを a.length つまり整数で割ると、整数除算 (切捨て除算) になってしまう。そこで、どちらかの値を to_f で実数に変換して、それから除算すれば実数除算になるというわけだ。結構気遣いが大変ですよね。

このように、手続きに「分かりやすい名前をつけて」おき、「既に作ったものを活用する」ことで新しく書く部分が短く読みやすくなるだけでなく、既にできている部分は手を入れないので間違いも入らないで済む。このあたりが手続きによる抽象化の利点ということである。

2.4 演習 6/7 — 素数の計算

素数については前回もやったが、それを配列を使ってもっと効率よく、という話だった。これもプログラムだけ示しておく。まず「見つかった素数を覚えておいてそれだけで割ってみる」方法。

```
def testmemo(n, a)
  a.each do |x|
    if n % x == 0 then return false end
  end
  return true
end

def primememo(n)
  a = []
  2.step(n) do |i|
   if testmemo(i, a) then puts(i); a.push(i) end
  end
end
```

配列 a は最初は空で、素数が見つかるごとに push で追加していく。

もう 1 つの方法は「エラトステネスのふるい」と呼ばれるアルゴリズム。

```
def primesieve(n)
  a = Array.new(n+1, true)
  2.step(n) do |i|
    if a[i]
      puts(i)
      i.step(n, i) do |j| a[j] = false end
    end
  end
end
```

配列 a を a[0]〜a[n] が使えるように大きさ n+1 で用意し、最初はすべて true(素数である) にしておく。次に、2 から始めて順に n 番目が「素数である」なら、それは素数だから打ち出すとともに、n の倍数 (2 倍、3 倍、…) はすべて false(素数でない) に変更する。

ところで、上のコードはまだ効率よくできる。それは、配列に false を入れていく動作は i が n の平方根のところまでやれば十分で、そこから先は印のついている番号を打ち出すだけでよい。つまり、r = Math.sqrt(n).to_i (最後の to_i というのは実数を切捨てて整数にするメソッド) としておき、ループを 2.step(r) と r.step(n) に分けて後半では配列の印つけを省略する。さらに、全ての数をチェックするのではなく、2 を最初に出力して後は 3 以上の奇数をチェックする改良も可能である。

2.5 再帰関数

関数の興味深い用法として、ある関数の中から直接または間接に自分自身を呼び出す、というものがある。これを**再帰** (recursion) という。たとえば、正の整数 x、y について、その最大公約数は次のようにして定義できたことを思い出そう。

$$gcd(x,y) = \begin{cases} x & (x = y) \\ gcd(x-y, y) & (x > y) \\ gcd(x, y-x) & (x < y) \end{cases}$$

これにそのまま従って Ruby の関数を書くことができる。例題全体を示そう。

```
def gcd(x, y)
  if x == y
    return x
  elsif x > y
    return gcd(x-y, y)
  else
    return gcd(x, y-x)
  end
end
```

プログラムそのものは大変分かりやすいでしょう? しかしこれでなぜ「堂々めぐり」にならずに計算が終了するのだろうか。それは、図 1 のようなものを描いてみれば分かる。

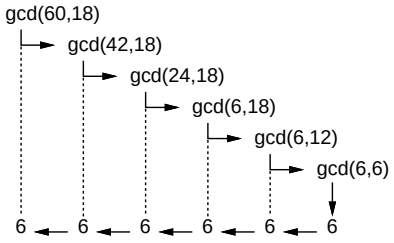

図 1: 再帰関数による最大公約数の計算

再帰関数 (再帰手続き) を作る時は、必ず次の原則に従う。

- 問題の「簡単で明らかな場合」は、すぐに答えを返す (上の例では $x = y$ の場合)。
- それ以外の場合は、問題を「少し簡単な問題に変形した上で」自分自身を呼び出す (上の例では、大きい方から小さい方を引くことで少し簡単にしている)。

これがうまくできていれば、堂々めぐりにならずに正しく実行できる。

演習 1 上の例題をそのまま打ち込んで動かせ。また、x や y に 0 や負の整数を入れるとどうなるかまず予測し、その後実際にやってみて予測を確認せよ。

演習 2 次のような再帰的定義に従った計算を再帰関数として書いて動かせ (関数部分以外は上の例題を直せば済む)。また、典型的な実行のようすを表す、図 1 のような図を描いてみよ (すみませんが何とか文字だけで工夫してみてください)。

a. 階乗の計算。

$$fact(n) = \begin{cases} 0 & (n = 0) \\ n \times fact(n-1) & (otherwise) \end{cases}$$

b. フィボナッチ数。

$$fib(n) = \begin{cases} 1 & (n = 0 \ or \ n = 1) \\ fib(n-1) + fib(n-2) & (otherwise) \end{cases}$$

c. 組み合わせの数の計算。

$$comb(n, r) = \begin{cases} 1 & (r = 0 \ or \ r = n) \\ comb(n-1, r) + comb(n-1, r-1) & (otherwise) \end{cases}$$

d. 正の整数 n の 2 進表現。

$$binary(n) = \begin{cases} \text{"0"} & (n = 0) \\ \text{"1"} & (n = 1) \\ binary(n \div 2)\text{+"0"} & (n \text{ が 2 以上の偶数}) \\ binary(n \div 2)\text{+"1"} & (n \text{ が 2 以上の奇数}) \end{cases}$$

この場合、関数の返す型は `String` であることに注意。また+は文字列の連結演算、÷は整数の除算 (切捨て除算) を表していることに注意 (Ruby では整数どうしの「/」は自動的に切捨て除算になる)。

3 レコード型と画像

3.1 レコード型の利用

前回説明したように、配列が「同じ型 (種類) の値が並んだもので、添字 (番号) により要素を指定する」のに対し、レコードは「違う型 (種類) の値でもよい、複数の値が組み合わさったもので、どの値 (フィールド) かは名前で指定する」ようなものだった。

Ruby ではレコード型は `Struct.new` を使ってまずレコードクラスを定義し、その後そのレコードクラスを使って個々のレコード (データ) を作る。具体的には、レコードクラスの定義は次のようにすればよい (レコード名は大文字で始まらなければならない)。

```
レコード名 = Struct.new(:名前, :名前, ...)
```

ここで「:名前」は前回やった記号 (シンボル) だけれど、まあこういう風に指定すると思っておけばよい。指定した名前はそれぞれがレコードのフィールドになる。個々のレコードを作るのは次による。

```
p = レコード名.new(値, 値, ...)
```

これによりレコード型の値が作られ、指定した値が各フィールドの初期値になる (順番はレコード定義の時に指定した順)。上の例ではそのレコードを変数 p に入れている。

たとえば、コンピュータ上で画像を扱うときは、多数の点の集まり (ピクセル) として扱うこと、そして各ピクセルの色は赤 (R)/緑 (G)/青 (B) の強さを 0〜255 の範囲の整数で表す方法が多いことはご存じと思う。このピクセルの情報を表すレコードを定義してみる:

```
Pixel = Struct.new(:r, :g, :b)
```

実際にこのレコードを使うときは、次のようになる:

```
p1 = Pixel.new(255, 255, 255) // RGB とも 255 の値
```

Ruby では配列のときと同様、レコードも new を使って作り出さないと使えないのに注意。

さらにレコードの場合も配列と同様、レコード本体はどこか「別の場所」に取られ、変数はそこを「指している」ことに注意。つまり、変数 p は最初は何も指していない状態 (図 2 上) で、代入によってレコードを指す (図 2 下)。なお、この「何も指していない状態」はレコード、配列、オブジェクト型の変数すべてに存在し得る状態で、そのときはそれらの変数には `nil` という特別な値が入っている。

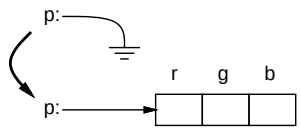

図 2: レコードの割り当て

3.2 2 次元配列

ピクセル 1 個ではつまらないので、ピクセルが 2 次元に (縦横に) 並んだ配列、つまり 2 次元配列 (添字が 2 個の配列) を作って、適当なサイズ、たとえば 200 × 300 の画像を表すことにする。そこで最初に 2 次元配列について説明しておく。

```
$img = Array.new(200)
```

とすると、要素数 200 の 1 次元配列ができ、その配列を変数`$img` に入れることになる。添字としては`$img[0]`〜`$img[199]` が使える。2 次元配列ということは、**この各要素が配列**であればよい (図 3)。

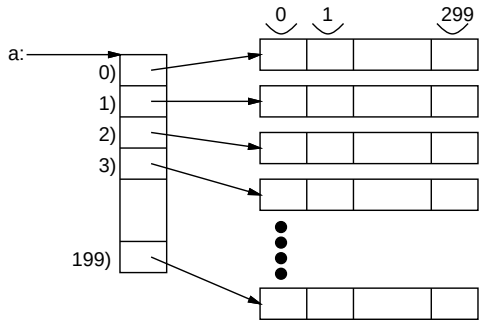

図 3: 2 次元配列

このような構造を作るには、要するに最初の 1 次元配列の各要素に大きさ 300 の配列を格納すればよい。とすると、次のようなコードになる。

```
$img = Array.new(200)
200.times do |j|
  $img[j] = Array.new(300)
  (以下略)
```

この 2 次元配列 (正確に言えば「配列の配列」だが) の各要素を指定するのには「`$img[10][25]`」のように 2 つ添字を指定することになる (2 次元目の添字の範囲は 0〜299 ということになる)。これで数学でいう x_{ij} のような 2 つの添字つき変数 (行列の要素など) ができたわけだ。

ところでここでレコードの話題に戻るが、今はこの 2 次元配列の各要素が RGB 値を持ったピクセルなので、2 次元配列を作ったらすぐにその各要素にレコード値を入れることにする。

```
$img = Array.new(200)
200.times do |j|
  $img[j] = Array.new(300)
  300.times do |i| $img[j][i] = Pixel.new(255,255,255) end
  (以下略)
```

なお、(255,255,255) というのは赤緑青とも最大に光った色、つまり「真っ白」を意味している。

3.3 例題: 画像を生成し書き出す

以下に示す例題は、色のついた円が2個重なっている画像を生成し書き出すものである (図 4)。説明の都合上、4つに分けて示しているが、もちろんこれらをくっつけた形でファイルに書いて読み込めばよい。

図 4: プログラムで生成した画像

```
Pixel = Struct.new(:r, :g, :b)

def initimage
  $img = Array.new(200)
  200.times do |j|
    $img[j] = Array.new(300)
    300.times do |i| $img[j][i] = Pixel.new(255,255,255) end
  end
end
```

まず最初に、レコード Pixel の定義がある。次は initimage だが、これは前節で説明した通り、2次元配列を用意して、その各要素にピクセルを表すレコード (真っ白に対応) を入れている。

画像の初期化に続いて、画像をファイルに書き出すメソッド writeimage を説明してしまおう。画像ファイルの形式には色々なものがあるが、ここでは、できるだけ出力が簡単な形式として PPM 形式の画像を出力するようにした。writeimage の引数として、作成するファイルの名前を渡すものとしている。

```
def writeimage(name)
  open(name, "w") do |f|
    f.puts("P3 300 200 255")
    200.times do |j|
      300.times do |i|
        p = $img[j][i]; f.puts("#{p.r} #{p.g} #{p.b}")
      end
    end
  end
end
```

open は指定された名前のファイルを用意し、その後ろのブロック内でそのファイルの読み (モード"r"のとき)/書き (モード"w"のとき) をできるようにしてくれる。ブロック内では、open から渡されて来るパラメタ f にファイルを読み書きするための File オブジェクトを使用すればよい。ここではファイルに書くためのメソッド puts だけを使っている。具体的には、(1)PPM ファイルの先頭部分として「色つき PPM で、サイズ 300 × 200、各色の最大値は 255」を表す情報を書き出し、続いて (2) 各ピクセルの RGB 値を空白で区切って順次書いて行く。

さて、初期化していきなりファイルに書いたら「真っ白」な画像ができるだけでつまらないので、何らかの「絵」を作成することにする。ここでは簡単な例として、「中心の XY 座標、半径、RGB 値」を指定して円を塗りつぶすメソッド `fillcircle` を作る。

```
def fillcircle(x, y, rad, r, g, b)
  200.times do |j|
    300.times do |i|
      if (i-x)**2 + (j-y)**2 < rad**2
        $img[j][i].r = r; $img[j][i].g = g; $img[j][i].b = b
      end
    end
  end
end
```

アルゴリズムは簡単で、「画像中のすべての点点 (i, j) について、それが中心 (x, y)、半径 rad の円内に入っているかどうか調べ、入っている時だけ指定された色をその点の R/G/B 値として書き込む」だけである。円の中に入っているかどうかはもちろん、ピタゴラスの定理により $(i-x)^2 + (j-y)^2 < rad^2$ で判断できる。

なお、ここで使っている表現は「配列の 2 つの添字のうち 1 番目が Y 座標、2 番目が X 座標で、しかも画像の下の方ほど Y が大きい」という奇妙なものだが、コンピュータグラフィクスではなぜかこうすることが多い (図 3 もこれに合わせて描いてある)。

以上で「材料」が揃ったので、画像を初期化し、円を 2 つ描き、ファイルに画像を書き出す」メソッド `mypicture` を用意した。

```
def mypicture
  initimage
  fillcircle(150, 30, 60, 255, 100, 70)
  fillcircle(190, 100, 30, 100, 200, 80)
  writeimage("t.ppm")
end
```

これを実行するとファイル `t.ppm` に PPM 形式の画像ファイルが作成される。PPM 形式の画像は ImageMagick というツールパッケージに含まれているコマンド `display` で表示できる。自宅でこの演習をやる場合には、次のところの説明を読んで画像表示ソフトを取り寄せてください: Windows 版を取りよせてください:

```
http://www.vector.co.jp/soft/dl/mac/art/se302845.html (Win)
http://mechanics.civil.tohoku.ac.jp/soft/node43.html  (Mac)
```

演習 3 画像ファイルを生成する例題を打ち込んでそのまま動かせ。動いたら円の位置や色を変更したり、円をもっと増やしてみよ。

演習 4 画像ファイルを生成する例題に、次のような手続きを追加してみよ。追加したものを活用した画像を生成してみること。

a. 長方形、楕円、三角形、直線を描く (指定した色で塗り潰す) ような手続き。長方形や楕円は回転できると嬉しいかも知れない。

b. 上記 a. や円の手続きについて、色を「重ね塗り」できるようにする。つまり透明度 $0 \leq p < 1$ を指定し、各 R/G/B 値について単に新しい値で上書きする代わりに $p\, r_{old} + (1-p)\, r_{new}$ のように混ぜ合わせた値にする。

c. 上記 a. や円の手続きについて、全範囲を均一な色で塗るのではなく、徐々に色調が変わって行くようにする。

d. その他、美しい絵を描くのにあるといいと思うもの。

演習 5 「円を塗りつぶす例題」はあまり効率がよくない (遅い)。速度を改善してみよ。演習 4 で作成したプログラムについても同様に効率を改善してみよ。

演習 6 「美しい絵」を生成するプログラムを作れ。何が美しいかの定義は各自に任されるものとします。

A　本日の課題 4A

「演習 2」で動かしたプログラムを含む小レポートを今日中に久野までメールで送ってください。

1. Subject: は「Report 4A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 「演習 2」で動かしたプログラムどれか 1 つのソース。
4. 以下のアンケートの回答。

   Q1. 手続き/関数について学びましたが、納得しましたか。

   Q2. 画像の生成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。

B　次回までの課題 4B

次回までの課題は「演習 2〜5」の (小) 課題 (4A で提出したものは除く) から 1 つと、「演習 6」との合計 2 つとします。演習 6 のプログラムが生成する画像についてはこのクラスの Web に掲示して皆様で相互に観賞して頂こうと思いますので、公序良俗に反する (ネットに掲示できない) 画像を生成するのはやめてください。そしてそろそろ、アイデア次第で「◎」をつけようと思いますので、4B は安易に出してしまわないことを勧めます (が、いいアイデアを思い付いたら他人が同じことをやって出してしまう前に出した方がいいことも確かです)。

各課題のために作成したプログラムはすべてレポートに掲載してください。また、今回は特に「どういう考えでこの画像を作ったか」の考察が重視されますのでよろしく。レポートは授業開始時刻の **10 分前**までに、上記と同様に久野までメールで送付してください。

1. Subject: は「Report 4B」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 演習 2〜5 で作成しだプログラムのソース。
4. その説明と分析/考察。
5. 演習 6 で作成したプログラムのソース。
6. その説明と分析/考察。
7. 下記のアンケートの回答。

   Q1. 手続きが使いこなせるようになりましたか。

   Q2. 思ったような画像を生成できましたか。できた/できなかったとすれば、どこがポイントでしたか。

   Q3. 課題に対する感想と今後の要望をお書きください。