第3章　条件分岐と繰り返し

この章では数値以外のデータとして**真偽値**と**文字列**を紹介する。実用的なプログラムは、複雑な判断を行いながら処理を行うことが多いが、この判断部分の計算に使われるのが**真偽値**である。また、利用者とのデータのやりとりのために文字情報を扱うことは多いが、そこで用いられるのが**文字列**である。

この章ではまた、沢山のデータを処理する方法の手始めとして、配列に対する**繰り返し**を紹介する。繰り返しによって規則的な画像を生成することや、統計処理など様々な処理を行うことができる。

3.1　条件分岐

3.1.1　場合分けを使った計算

場合分け、つまり「ある場合にはこの値、別の場合にはこの値」といった計算を行いたいことは少なくない。例えば「2人の得点 x, y の最高点 $\max(x, y)$」を求める問題は「$y < x$ の場合は x, そうでなければ y」という計算だと言える。他にも「x の絶対値 $\mathrm{abs}(x)$」($x > 0$ かどうか) や「2次方程式 $ax^2 + bx + c = 0$ の実数解の個数 $\mathrm{solutions}(a, b, c)$」(判別式の正負) のような計算もやはり場合分けである。

このような場合分けは、数学では次のように書くことができる。

$$\max(x, y) = \begin{cases} x & (y < x \text{ のとき}) \\ y & (\text{それ以外}) \end{cases}$$

プログラミング言語では**条件分岐**を使って、場合分けのある計算を表わす。例えば max であれば次のように定義できる。

上の定義の 2 行目から 6 行目までが場合分けをしている 1 つの式である。`if`, `else`, `end` が区切りとして使われているが、数学的な書き方との対応は自明だろう。

以下は実際に使ってみた様子である。

```
1 irb(main):003:0> load("./max.rb")
2 => true
3 irb(main):004:0> max(123, 456)
4 => 456
```

```
1 def max(x,y)
2   if y < x
3     x
4   else
5     y
6   end
7 end
```

ファイル 3.1: max.rb

```
5 irb(main):005:0> max(max(12, 34), max(56, 78))
6 => 78
```

練習 3.1 (絶対値) 数値 x の絶対値を求める関数 abs(x) を定義せよ。

3.1.2　3 通りの場合分け

ある値 x の符号、つまり x の正負に応じて $1, -1, 0$ のどれかを答えるような関数 $\mathrm{sign}(x)$ を考えよう。数学であれば次のような 3 つの場合分けによって書ける。

$$\mathrm{sign}(x) = \begin{cases} -1 & (x < 0) \\ 1 & (0 < x) \\ 0 & (\text{それ以外}) \end{cases}$$

条件分岐は 1 つの条件を調べ、それが成り立つ場合とそうでない場合に分けるのが基本なので、3 つの場合に分けるには**入れ子**にする必要がある。つまり、条件分岐によって場合分けされた式の中に、さらに条件分岐を書く必要がある。

(章末の練習問題: 3.5)

3.1.3　複雑な条件

現実世界の問題は、かなりこみ入った場合分けを行わなければいけないこともある。原理的には if y < x という形の条件分岐だけを何重にも入れ子にすることで、かなり複雑な場合分けを行うことも可能ではある。しかし、そうやって出来あがった条件分岐がどんな場合分けをしているのかを理解することは簡単でなくなってしまう。

そのため Ruby をはじめとする多くのプログラミング言語には、色々な種類の比較を行うための記号と、条件式を組み合わせるための記号が用意され

```
1  def sign(x)
2      if x < 0
3          -1
4      else
5          if 0 < x
6              1     # not(x<0) and 0<x
7          else
8              0     # not(x<0) and not(0<x)
9          end
10     end
11 end
```

ファイル 3.2: sign.rb

ており、複雑な条件を 1 つの条件式として書けるようになっている。より詳しくは第 3.2 節で説明するが、次のようなものが代表的である。

色々な比較

数の大小および、等しさの比較を行う等号と不等号には次のようなものがある。いずれも数学における等号・不等号に似せている。

書き方	数学	意味
`x > y`	$>$	x が y より大きい
`x >= y`	$\geq$	x が y 以上
`x == y`	$=$	x と y が等しい（x=y でないことに注意）
`x < y`	$<$	x が y より小さい
`x <= y`	$\leq$	x が y 以下
`x != y`	$\neq$	x と y が異なる

条件式の組み合わせ

条件式を組み合わせてより複雑な条件式を作ることができる。Ruby では ||, &&, ! という記号を使って「A または B」「A かつ B」「A でない」という条件を表わすことができる。以下に例を示す。

書き方	意味
`x > y || x == 0`	`x > y` または `x == 0`
`x < y && y < z`	`x < y` かつ `y < z`
`!(x < y && y < z)`	`(x < y` かつ `y < z)` でない

(章末の練習問題: 3.6)

3.2　真偽値を与える論理演算

第 3.1 節では `if y < x` のように、数の大小などに応じて場合分けをする方法や、複数の条件を `&&` などで組み合わせることで、より複雑な場合分けができることを見た。ここでは、そこで使われた条件式を、**真偽値**に関する**論理演算**として整理しよう。

まず**真偽値**とは、**真**つまり「正しい」と**偽**つまり「正しくない」という 2 つの値のことである。Ruby では真を表わす値を `true`, 偽を表わす値を `false` と書く。

条件式 `1 < x` は、2 つの数値を比較して真偽値を求める式である。数式 `1 + x` が 2 つの数値の合計値を求める式であることと対比させると、結果の値の種類が違うことを除けばよく似ている。実際、`1 < x` のような式を単独で計算することもできる。

```
irb(main):003:0> x = 3
=> 3
irb(main):004:0> 1 < x
=> true
irb(main):005:0> x == 2
=> false
```

また、答えが真偽値となるような関数を定義することもできる。例えば「ある数 x が偶数である」ことは、x を 2 で割った余りが 0 かどうかで判定できるので、次のような関数として定義することができる。

```
def is_even(x)
  x%2 == 0
end
```

ファイル 3.3: `is_even.rb`

このように定義した関数は、条件式の中で使うことができる。例えば、次のような関数 $\mathrm{tnpo}(n)$ を考える。

この関数名は、奇数の場合の式の英語読み “three n plus one” からとっている。

$$\mathrm{tnpo}(n) = \begin{cases} \frac{n}{2} & (n \text{ が偶数}) \\ 3n+1 & (n \text{ が奇数}) \end{cases} \tag{3.1}$$

これを Ruby の関数として定義する際には、「`n` が偶数」かどうかを判定する必要が生じるが、上で定義した `is_even(x)` を使えば次のように定義できる。

論理演算子`&&`, `||`, `!` (第 3.1.3 節) は、1 つまたは 2 つの真偽値から真偽値を求める演算である。これらを使った式も単独で計算できる。

```
irb(main):007:0> true && 1 < x
```

```
load("./is_even.rb")

def tnpo(n)
  if is_even(n)
    n/2
  else
    3*n + 1
  end
end
```

ファイル 3.4: tnpo.rb

```
=> true
irb(main):008:0> false || true && !(false||true)
=> false
```

条件を組み合わせた論理式は、一見して何を判定しているのかが分かりづらいので、関数として定義し、意味を明確にすることは重要である。例えばある数 x が 0 に近いことを判定する near_zero(x) は次のように定義できる。
(章末の練習問題: 3.7 3.8)

3.3　文字列

名前や文章のような情報は、プログラム中では文字を 1 列に並べた**文字列**として扱われる。文字列は、各文字の**文字コード** に対応する整数が沢山並んでいるものなので、配列の一種とも言える。実際、「先頭から n 文字目を参照する」とか「長さを調べる」といった配列と同様の操作が用意されている。

「情報」(川合編, 東京大学出版会)p.19 参照。

本書では簡単のために英語のアルファベット、数字、記号のいわゆる ASCII 文字集合だけからなる文字列のみを扱う。以下、具体例を用いて文字列の扱いを簡単に説明する。

例として「abra」と「cadabra」という言葉を操作してみよう。このような言葉を値として扱うためには二重引用符 (") 記号で囲めばよい。

実はこの書き方は load 命令に与えるファイル名のところですでに見ている。

関数 tnpo(n) は練習 4.12c, 練習 4.8c で紹介する Collatz 予想に登場する計算である。tnpo(7), tnpo(tnpo(7)), tnpo(tnpo(tnpo(7))), ... を計算してみよ。

```
1  def epsilon()
2    0.0000001
3  end
4
5  def near_zero(x)
6    -epsilon() < x && x < epsilon()
7  end
```

ファイル 3.5: near_zero.rb (この定義では充分に小さな値 ε として定数関数 (第 1.5.4 節) epsilon() を定義して使っている。)

```
1  irb(main):003:0> s = "abra"
2  => "abra"
3  irb(main):004:0> t = "cadabra"
4  => "cadabra"
```

これで変数 s と t に「abra」と「cadabra」という文字列がしまわれた。2 つの**文字列をつなげる**には、+ 記号を使う。

```
5  irb(main):006:0> u = s + t
6  => "abracadabra"
7  irb(main):007:0> "123" + "456"
8  => "123456"
```

数値の足し算と同じ記号を使っているが、文字列の場合はつなげた文字列を作る。従って 7 行目のように 123 という**文字列**に 456 という**文字列**を+した結果は 123456 という文字列になる。

文字列処理では、色々な長さの文字列を扱うことが多い。例えば、名簿にある名前の長さは人それぞれである。そのような場合には、それぞれの文字列の長さを調べ、その長さに応じた処理をすることになる。

文字列の長さを調べる文字列を表わす式の後に.length() と書く。

```
9  irb(main):009:0> s.length()
10 => 4
11 irb(main):010:0> (s + t).length()
12 => 11
```

文字列の一部分を取り出すには次のようにする。

```
13 irb(main):012:0> s[0..0]
14 => "a"
```

```
15 irb(main):013:0> s[1..2]
16 => "br"
```

ここで「..」の前後は、取り出す最初の文字の添字と最後の文字の添字を意味する。

(章末の練習問題: 3.9)

3.4　繰り返しによる画像の作成

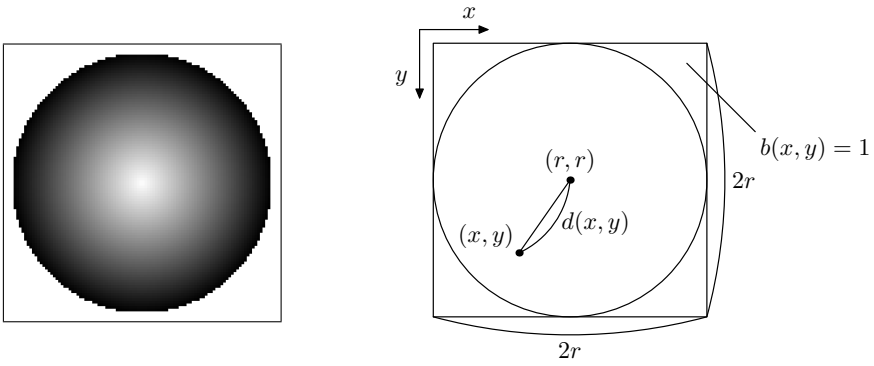

図 3.1: 繰り返しによって作成した画像 (左) とその位置関係 (右)

図 3.1(左) のような画像を作ることを考えよう。一見難しそうに思えるかもしれないが、各点について画像の中心からの距離を求め、それに従って明度を決めているだけなので、かなり規則的なものである。具体的には、図右のように画像の大きさを $2r \times 2r$, 座標 (x, y) の点 (r, r) からの距離を $d(x, y)$ としたときに、点 (x, y) の明度 $b(x, y)$ を以下のように決めている。

$$b(x,y) = \begin{cases} \frac{r-d(x,y)}{r} & (d(x,y) \le r) \\ 1 & (d(x,y) > r) \end{cases} \tag{3.2}$$

問題は、どうやって大きさ $2r \times 2r$ の配列を作り、中身を上の計算式で定めてやるかである。このためには**繰り返し**命令が必要になる。順にやり方を見てゆこう。

3.4.1　与えられた大きさの 1 次元配列を作る

これまで配列を作る場合は、 `[1,2,3]` のような式を書いていた。この場合、`[ ]` の内側の式の個数がそのまま配列の大きさになった。この方法では、予め決められた大きさの配列しか作れない。

与えられた大きさの配列を作るには Ruby では `Array.new(10)` のような式を書く。10 の部分は任意の式で置き換えることができる。

> 注意:　以降では作成した配列を画像として表示するので、`irb` ではなく第 2.1 節で説明した `isrb` を使う。もし `irb` を使っている場合は一度終了して `isrb` を起動しなおしておくこと。

```
1 >> image = Array.new(6)
2 => [nil, nil, nil, nil, nil, nil]
3 >> a = Array.new(3)
4 => [nil, nil, nil]
```

作られた配列は `[1,2,3]` のような式によって作られたものと同様に扱える。ただし、作られた当初の中身は `nil` という「空である」ことを示す特別な値になっている。

3.4.2　繰り返しによって、配列の全要素をそれぞれ変更する

次に配列の要素を規則的に決める方法である。数学風には

$$a_i = 0 \qquad (\forall i \in 0 \ldots 2)$$

と書いて「数列 a の要素 a_0 から a_2 までがすべて 0 である」ことを表わすが、Ruby ではこれに似た `for` 命令を用いて、配列 `a` の全要素を 0 に変更できる。

```
5 >> for i in 0..2
6 >>   a[i] = 0
7 >> end
8 => 0..2
9 >> a
10 => [0, 0, 0]
```

上の画面の 5 行目から 7 行目は、「`i` を 0,1,...,2 の順に変化させて、毎回 `a[i]=0` を実行せよ」という**繰り返し命令** `for` である。この命令を実行した後の `a` を見ると (9 行目) 確かにすべてが 0 になっている。

練習 3.2 (0 ばかりの配列を作る) 大きさ `n` で中身がすべて 0 であるような 1 次元配列を作る関数 `make1d(n)` を定義せよ。(ヒント: 3 行目から 7 行目までがほぼそのまま関数定義の中身になる。)

```
1 >> make1d(10)
2 => [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

(章末の練習問題: 3.11)

3.4.3　2 次元配列を作る

2 次元の配列は、すべての中身が 1 次元の配列であるような 1 次元の配列なので、6 行 3 列の配列を作りたかったら練習 3.2 で作った make1d を使って次のようにすればよい。

```
>> for i in 0..5
>>     image[i] = make1d(3)
>> end
=> 0..5
>> image
=> [[0, 0, 0], [0, 0, 0], [0, 0, 0], [0, 0, 0], [0,
    0, 0], [0, 0, 0]]
```

練習 3.3 (2 次元配列) h 行 w 列の配列を作る make2d(h,w) を定義せよ。ただし、作られる配列の中身はすべて 0 にせよ。

```
>> make2d(2,3)
=> [[0, 0, 0], [0, 0, 0]]
```

3.4.4　2 重の繰り返し

2 次元の配列を作ったら、あとはすての x, y の組み合わせについて image[y][x] = b(r,x,y) のような代入をすれば図 3.1 のような画像を作ることができる。

このときすべての x, y の組み合わせについて代入をするためには、**2 重の繰り返し**を行う。次の関数定義 sphere を見てほしい。

```
def sphere(r)
  image = make2d(2*r, 2*r)
  for y in 0..(2*r-1)
    for x in 0..(2*r-1)
      image[y][x] = b(r,x,y)
    end
  end
  image
end
```

ファイル 3.6: sphere.rb

この定義では 3 行目から 7 行目までが y を変化させる繰り返しである。その内側の 4 行目から 6 行目 までは、x を変化させる繰り返しなので、

- y を 0 として、x を $0, 1, 2, \ldots, 2\mathtt{r}-1$ と変化させて 5 行目を実行
- y を 1 として、x を $0, 1, 2, \ldots, 2\mathtt{r}-1$ と変化させて 5 行目を実行
- y を 2 として、x を $0, 1, 2, \ldots, 2\mathtt{r}-1$ と変化させて 5 行目を実行

$\vdots$

- y を $2\mathtt{r}-1$ として、x を $0, 1, 2, \ldots, 2\mathtt{r}-1$ と変化させて 5 行目を実行

という順序で実行が起き、結果として画像のすべての点の明度を決めることができる。

ここでの 5 行目の右辺は p. 37 の式 3.2 (に対応する関数 b) であるが、この式を適当なものに変えることで色々な図形を作成することができる。

練習 3.4 (繰り返しによる画像の作成) 式 3.2 の計算をする関数 `b(r,x,y)` を定義せよ (x, y だけでなく r も引数となっていることに注意せよ)。`show(sphere(20))` を実行して図 3.1 のような画像が表示されることを確めよ。

(章末の練習問題: 3.12 3.13)

3.5　定義のまとめ

条件分岐: 式$_1$ が成り立つときは 式$_2$ を、そうでないときは 式$_3$ を計算する条件分岐は次のように書く。このとき 式$_1$ を**条件式**という。

```
if 式1
    式2
else
    式3
end
```

真偽値: `true` と `false` はそれぞれ、真と偽を表わす値である。

文字列を作る: 2 つの " 記号の間にある文字や記号は文字列を作る。

文字列の結合: 式$_1$ + 式$_2$ という式は、式$_1$, 式$_2$ の値が文字列のとき、それらの文字列をつなげた文字列を作る。

文字列の長さ: 式$_1$`.length()` という式は、式$_1$ が表わす文字列の長さを求める。

部分文字列: 式$_1$`[`式$_2$`..`式$_3$`]` という式は、式$_1$ が表わす文字列の 式$_2$ 番目から 式$_3$ 番目までを取り出した文字列を作る。

空の値: `nil` は「無い」ことを表わす特別な値である。

配列の作成: `Array.new(`式`)` という式は、大きさが式の値であるような配列を作る。作られた配列の中身はすべて `nil` である。

繰り返し: 変数の値を式$_1$の値から式$_2$の値まで 1 ずつ順に変化させながら、命令$_1$から命令$_n$を毎回実行する**繰り返し**は次のように書く。

```
for 変数 in 式1..式2
  命令1
  ⋮
  命令n
end
```

3.6　章末問題

練習 3.5 (条件分岐) 次のような計算を行う関数を定義せよ。

a) 2 次方程式 ax^2+bx+c $= 0$ の実数解の個数を求める `solutions(a,b,c)`.
判別式の値だけでなく、1 次方程式になっている場合も考えよ。

b) 2 次方程式 ax^2 + bx + c $= 0$ の実数解を 1 つ求める `solve1(a,b,c)`.
判別式の値だけでなく、1 次方程式になっている場合も考えよ。

c) 3 つの異なる値 x, y, z が与えられたときの中央値を求める `median(x,y,z)`.
中央値とは大きさ順に並べたときに真ん中に来る値のことである。

d) 年間所得が `income` 円だったときの所得税の税額を求める `income_tax(income)`.
ただし 2008 年の時点での日本の所得税率は次のようになっている。

(a) 195 万円以下の所得に対して 5%.

(b) 195 万円より多く 330 万円以下の所得に対して 10%.

(c) 330 万円より多く 695 万円以下の所得に対して 20%.

(d) 695 万円より多く 900 万円以下の所得に対して 23%.

(e) 900 万円より多く 1,800 万円以下の所得に対して 33%.

(f) 1,800 万円より多い所得に対して 40%.

ただし、上の税率は超過分についてのみ適用される。つまり、所得が 250 万円だった場合は、そのうちの 195 万円に対して 5%, 残りの55万円に対して10%の税が課されるので、$195 \times 0.05 + (250 - 195) \times 0.10 = 15.25$(万円) が税額となる。

東大生の家庭のうち約 19%は、950 万円から 1050 万円の年収を得ているという。収入と所得が同じだったとして、年間所得 1000 万円の場合の税額を計算してみよ。

学生生活実態調査 (2006 年・東京大学) の結果より。

e) 西暦 year 年 2 月の日数を求める days_of_february(year). ただし、year は次の条件を満たすときは閏年であることに注意せよ。

   (a) year が 4 の倍数である場合は閏年である

   (b) 条件 (a) を満たしていても、year が 100 の倍数の場合は閏年でない

   (c) 条件 (b) を満たしていても、year が 400 の倍数の場合は閏年である。

f) 西暦 year 年 month 月の日数を求める days_of_month(year,month).

練習 3.6 (条件式の組み合わせ) 練習 3.5c で定義した関数 median を定義し直して、(a)if y < x という形の条件式だけ使った median1 と、(b) && などを使って条件式を組み合て条件分岐の数を最小にした median2 を作れ。どちらの定義が分かりやすいだろうか。

練習 3.7 (真偽値を求める関数) 次のような計算を行う関数を定義せよ。

a) x が y で割り切れることを判定する divisible(x,y).

b) x < y < z であることを判定する ascending(x,y,z)

c) 西暦 y 年が閏年であるときに真となる leap_year(y).

   ただし現行の閏年の条件は「西暦年が 4 で割り切れる年を閏年とする。但し西暦年が 100 で割り切れるものの中でさらに 4 で (100 で割った) 商が割り切れない年は平年とする」というものである。従って 1900 年は平年、2000 年は閏年、2008 年も閏年、2100 年は平年である。

d) 添字 i が配列 a の範囲内かどうかを判定する within_range(a,i). ただし配列 a の添字は 0 からはじまり、a.length() の値未満である。

e) 座標 (x, y) が濃淡画像 img の中にあるかを判定する within_image(img,x,y).
   (ヒント: 上で定義した within_range が使える。)

練習 3.8 (論理関数) 真偽値を与えて真偽値を答えるような関数は**論理関数**という。次のような真理値表の通りに答えを返す論理関数 xor(x,y), implies(x,y) を定義せよ。

「情報」(川合編, 東京大学出版会)p. 158 参照。

x	y	xor(x,y)	implies(x,y)
false	false	false	true
false	true	true	true
true	false	true	false
true	true	false	true

練習 3.9 (文字列) 次のような関数を定義せよ。

a) 2 つの文字列 s, t のうち、長い方の文字列を求める longer(s,t).

b) 長さ 2 以上の文字列 s の先頭と最後の文字を取り除いた文字列を作る trim(s). 例えば trim("abc")⇒"b" となる。

c) 文字列 s の前半と後半を入れ替えた文字列を作る upsidedown(s). 例えば upsidedown("takeover")⇒"overtake" となる。余力があれば奇数文字のときに upsidedown("least")⇒"stale"となるように定義してみよ。

練習 3.10 (条件分岐を使った配列の操作)　a) a は 1 次元の配列とする。a の x 番目とその前後の要素数を数える関数 length3(a,x) を作れ。

$$\texttt{length3}(\texttt{a},\texttt{x}) = \begin{cases} 3 & (0 < \texttt{x} < \ell - 1) \\ 1 & (\texttt{x} = 0 \text{ かつ } \ell = 1) \\ 2 & (\ell > 1 \text{ かつ } (\texttt{x} = 0 \text{ または } \texttt{x} = \ell - 1)) \end{cases}$$

ただし ℓは配列 a の長さ

注: この関数は次に定義する sum3(a,x) で合計する要素の個数を求めている。

b) 練習 2.2 で定義した image_average9(image,x,y) および練習 2.3 で定義した sum3(a,x), array_average3(a,x) を、x(や y) の値が配列の端だった場合でも正しく計算できるように定義を修正した関数image_average(image,x,y), sum, array_average(a,x) を定義せよ。例えば a が [1,2,3,4] のとき sum3(a,3) は 3 と 4 の平均である 3.5 と答える。

c) 配列 a の i 番目と j 番目の要素を入れ替える関数 swap(a,i,j) を作れ。例えば a = [0,1,2,3]↵ swap(a,1,3) の順に命令を実行した場合、a は [0,3,2,1] となる。

d) 配列 a の i 番目と i + 1 番目の数値の大きさを比べ、前者が後者より大きいときに両者を入れ替える関数 swap_ascending(a,i) を作れ。

例えば a = [3,1,4,1]↵swap_ascending(a,0) の順に命令を実行した場合、a は [1,3,4,1] となる。さらにそこで swap_ascending(a,1) を実行しても a は [1,3,4,1] のままになる。

練習 3.11 (数列を作る) 大きさ n で i 番目の値が $i/\mathtt{n}$ になっているような 1 次元配列を作る関数 gradation(n) を定義せよ。(ヒント: make1d で 0 を代入している所を i を使った式に変える。整数どうしの割り算は整数商を求めてしまうことに注意せよ。)

練習 3.12 (色々な図形 *) 図 3.2 のような画像を作成する関数をそれぞれ定義せよ。

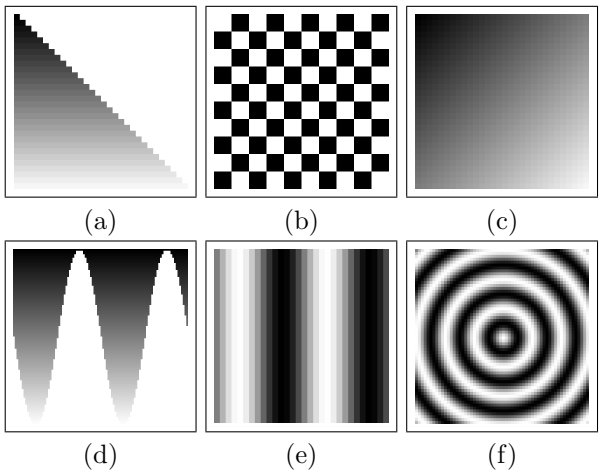

図 3.2: 繰り返しを使って作った色々な画像

練習 3.13 (画像の加工 **) 画像を作成する場合に、他の画像の情報を参照することで画像を加工することができる。例えば、濃淡画像 s を明るくするには、例えば各点の明度 b を $\frac{b+1}{2}$ にしてやればよいので、同じ大きさの画像 img を作り、各座標 (x, y) について以下のように定めてやればよい。

```
img[y][x] = (s[y][x]+1)*0.5
```

(配列の大きさは、s.length() で高さが、s[0].length() で幅が求められる。)

以下のような関数を定義し、練習 3.12 で作った画像に適用してみよ。

a) 与えられた画像 img を明るくした画像を作る関数 brighter(img).

b) 与えられた画像 img1, img2 を合成した画像を作る blend(img1,img2). ただし合成は、同じ座標の点の明度を平均化するものとする。

c) 与えられた画像 img を平均化する blur(img). (練習 2.2 で作った image_average9 を使う。)

配布プログラム utlogo.rb には、東京大学のロゴマークの白黒濃淡画像の配列を作る関数 logo() が定義されている。例えば上で定義した blur(img)

を使って `show(blur(logo()))`, `show(blur(blur(logo())))` とすると、次のようにぼやけた画像を作ることができる。

`logo()`

`blur(logo())`

`blur(blur(logo()))`