プログラミングプロジェクト＃2 – 分岐と反復

久野 靖 (電気通信大学)

2018.6.27

1　はじめに

前回は始めてのプログラミングなので一直線のコードで計算するだけでしたが、今回はいよいよ制御構造 (分岐や反復) を使っていただきます。今回の目標は次の通り。

- 基本的な制御構造 (分岐・反復) を理解し、これらを使ったプログラムが書けるようになる。
- 基本的な制御構造を用いたアルゴリズムやプログラムについて考えられるようになる。

今回から毎回、前回の演習問題から抜粋して解説を掲載します。この部分は自分で課題をやってみてから読むことを勧めます。

2　前回演習問題の解説

2.1　演習 3a — 四則演算を試す

演習 3a は和の計算でした。メソッド内の計算式を取り替えればいいだけなので簡単です。

```
def add(x, y)
  return x + y
end
```

動かしているところも見てみましょう。

```
irb> add 3.5, 6.8
=> 10.3
```

四則つまり和、差、商、積の場合も上と同じにやればいいわけですが、和、差、商、積のために 4 つメソッドを作る代わりに 1 つで済ませるという方法を考えてみましょう (半分くらいは新しい内容の紹介を兼ねています)。まず先に説明したように、メソッドの最後に値を返す代わりに、puts などで順次画面に書き出す方法があります。

```
def shisoku0(x, y)
  puts(x+y)
  puts(x-y)
  puts(x*y)
  puts(x/y)
end
```

動かしているところは次のとおり。

```
irb> shisoku0 3.3, 4.7
8.0
-1.4
```

```
15.51
0.702127659574468
=> nil
```

なるほど 4 つの値が順次打ち出され、最後に shisoku0 の結果としては **nil**(何もないことを示す値) が返されています。

上の方法だと「1 つの結果が返る」のでないのがちょっと、という気がするかもしれません。そこで次に、1 つの文字列を返し、その中に 4 つの数値が埋め込まれている、というふうにしてみましょう。

Ruby では**文字列** (string — 文字が並んだデータ) は「'...'」または「"..."」のように**シングルクォート**または**ダブルクォート**で囲んで表しますが、ダブルクォートのほうは内部に色々なものを埋め込む機能がついています。[1] 具体的には、文字列の中に「#{...}」という形のものがあると、中カッコ内の式を**評価** (evaluation — 値を計算すること) して、結果をそこに埋め込んでくれます。

これを利用した「四則演算」のメソッドを示します。

```
def shisoku1(x, y)
  return "#{x+y} #{x-y} #{x*y} #{x/y}"
end
```

実行しているところは次のとおり。

```
irb> shisoku1 3.3, 4.7
=> "8.0 -1.4 15.51 0.702127659574468"
```

確かに、「文字列」が打ち出されていて、その中に 4 つの数値が埋め込まれています。

もう 1 つ、Ruby など多くの言語では値の並んだものを**配列** (array) という機能で扱います。Ruby では [...] の中に値をカンマで区切って並べることで配列を直接書けるので、これを使って 4 つの数値をまとめて返すことができます。[2]

```
def shisoku2(x, y)
  return [x+y, x-y, x*y, x/y]
end
```

実行しているところは次のとおり。

```
irb> shisoku2 3.3, 4.7
=> [8.0, -1.4, 15.51, 0.702127659574468]
```

ここでは文字列の場合とあまり変わらない感じがするかもしれませんが、配列では返された値の中から「0 番目」「1 番目」など番号を指定して特定の要素を取り出せるので、より便利に使えます。

2.2 演習 3b — 剰余演算

演習 3b は剰余演算「%」を試すというものでした。演算子を取り換えるだけなので、プログラムは簡単ですね。

```
def jouyo(x, y)
  return x % y
end
```

実行してみましょう。

---

[1] ダブルクォートは埋め込み機能等のために**特殊文字** (special character — 英数字以外の文字) を様々に解釈します。そのようなことをせずに文字列をそのまま表示させたい場合はシングルクォートを使ってください。

[2] return の後だと囲んでいる [ ] を省略できますが、場所によっては省略できないので常に書くことを薦めます。

```
irb> jouyo 8, 5
=> 3
irb> jouyo 20, 5
=> 0
irb> jouyo -8, 5
=> 2
irb> jouyo -21, 5
=> 4
```

マイナスの時も試しましたか? 「ここでマイナスだとどうだろう」と気付くようになってください。

それで、マイナスの時も剰余は負にならず、つまり「5 間隔」というのがマイナスまでずっと続いている、というふうに考えるのでしょうね。では、割る数がマイナスだったらどうでしょうか?

```
irb> jouyo 8, -5
=> -2
irb> jouyo -8, -5
=> -3
```

つまり剰余の符号は割る数の符号と一致するようになっているわけです。[3]

演習 3c — 円錐の体積

演習 3c は円錐の体積でした。底面の半径 r、高さ h として、まず円錐の底面の面積は πr^2。体積はこれに高さを掛けて 3 で割ればできます。

```
def cornvol(r, h)
  return (r**2*3.1416*h) / 3.0
end
```

ちなみに「`**`」はべき乗の演算子です。もちろん 2 乗は「`r*r`」と書いても構いません。

```
irb> cornvol 3.0, 4.0
=> 37.6992
```

ところで「円周率が 3.1416 というのは不正確だ」と思う人もいそうですね。しかし、コンピュータ上の計算は「電卓での計算」と同様、有限の桁数でしか行えないのであり、自分で必要と思う適当な桁数を決めてその範囲でやるしかないので、有効数字 5 桁くらいでと思うならこれでよいわけです。[4]

2.3 演習 3d — 平方根

平方根は `Math.sqrt(`x`)` で計算できるので、とりあえずそれを出力します。ここでは `puts` を使って複数の値についてまとめて出力しましたが、1 個ずつ手で値を与えて試してももちろん構いません。

```
def sqrts1
  puts(Math.sqrt(2))
  puts(Math.sqrt(3))
  puts(Math.sqrt(5))
end
```

実行してみます。

[3] 剰余演算の振舞いは、プログラミング言語によって多少の違いが見られる箇所です。

[4] 3.141592653589793 くらいまでは扱える精度があるので、この定数をいちいち書くのは嫌だという人のために `Math::PI` と書いてもよいようになっています。同様に、自然対数の底 e は `Math::E` で表せます。

```
irb> sqrts1
1.4142135623730951
1.7320508075688772
2.23606797749979
=> nil
```

小数点以下 14〜16 桁表示されていますが、正しい値でしょうか。検索すれば調べられますが、正確な平方根の値を掲げておきます。

$$\sqrt{2} \approx 1.41421356237309504880168872420969807856 96$$
$$\sqrt{3} \approx 1.73205080756887729352744634150587236694 28$$
$$\sqrt{5} \approx 2.23606797749978969640917366873127623544 06$$

表示されている範囲では合っています。なお、表示する小数点以下の桁数を指定して表示することもできます。それには「`printf("%.`桁`g\n",`値`)`」という指定を使えばよいのです。20 桁でやってみましょう。

```
def sqrts2
  printf("%.20g\n", Math.sqrt(2))
  printf("%.20g\n", Math.sqrt(3))
  printf("%.20g\n", Math.sqrt(5))
end
```

実行してみます。

```
irb> sqrts2
1.4142135623730951455
1.7320508075688771932
2.2360679774997898051
=> nil
```

さっき表示されたより先の部分は正しくありません。つまり「おまかせ」の表示はおおむね必要な桁数だけ表示するようになっているといえます。ということで、精度としては 16〜17 桁程度ということになるでしょう。

3　基本的な制御構造

3.1　実行の流れと制御構造

ここまでに出てきたアルゴリズムおよびプログラムはすべて「1 本道」、つまり上から順番に実行して一番下まで来たらおしまい、というものでした。単純な計算ならそれでも問題ありませんが、手順が複雑になってくると、実行の流れをさまざまに切り換えていくことが必要になります。この、実行の流れ切り換える仕組みのことを、一般に**制御構造** (control structure) と呼びます。

制御構造の表現方法の 1 つに**流れ図** (flowchart) があります。流れ図では、図 1 にあるような「処理を示す箱」「条件による枝分かれを示す箱」などを矢線でつなげて実行の流れを表現します。流れ図は一見分かりやすそうですが、作成に手間が掛かる、場所をとる、ごちゃごちゃの構造を作ってしまいがち、という弱点のため、今日のソフトウェア開発ではあまり使われません (このため本資料でも、流れ図の代わりに擬似コードを主に用います)。

アルゴリズムを記述する時にはさまざまな実行の流れを組み立てますが、今日ではそれらの実行の流れは、図 1 に示す 3 つの制御構造を組み合せる形で作り出していくのが普通です。

- **順次実行**ないし**連接** — 動作を順番に実行していくこと。

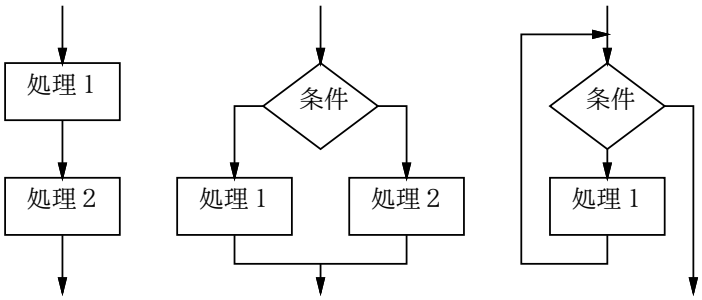

図 1: 3 つの基本的な制御構造

- **枝分かれ**ないし**分岐** — 条件に応じて 2 群の動作のうちから一方を選んで実行すること。
- **繰り返し**ないし**反復** — 条件が成り立つ限り一群の動作を繰り返し実行すること。[5]

なぜこの 3 つが基かというと、「どんなにごちゃごちゃの流れ図でも、それと同等の動作を、この 3 つの組み合わせによって作り出せる」という定理があり、そのためにこの 3 つさえあればどのような処理の流れでも表現可能だからです。連接については単に動作を並べて書いたものは並べた順番に実行される、というだけなので、以下では残りの 2 つの制御構造をコード上で表現するやり方と、それらを組み合わせてアルゴリズムを組み立てていくやり方を学びます。

3.2 枝分かれと if 文

上述のように、枝分かれとは、条件に応じて 2 群の動作のうちから一方を選んで実行するものです。擬似コードでは枝分かれを次のように書き表すものとします (「動作 2」が不要なら「そうでなければ」も書かなくてもかまいません)。

- もし ～ ならば、
- 　　動作 1。
- そうでなければ、
- 　　動作 2。
- 枝分かれ終わり。

Ruby ではこれを **if 文** (if statement) を使って表します (右側は「動作 2」のない場合です)。

```
if 条件 then
  ... 動作 1 ...
else
  ... 動作 2 ...
end
```

```
if 条件 then
   ... 動作 1 ...
end
```

`then` は Ruby では省略できますが、ただし「動作 1」を条件と同じ行に書く場合には省略できません。

「条件」については、当面は次の形のものがあると思っておいてください。

- 比較演算 — 「`x > 10`」等、2 値を比べるもの。**比較演算子**としては次の 6 種類がある。[6]

`>`	`>=`	`<`	`<=`	`==`	`!=`
より大	以上	より小	以下	等しい	等しくない

- 条件の組み合わせとして次が使える。[7] これらを「()」でくくったり複数組み合わせられる。

[5] 実行の流れを図示すると環状になるので、**ループ** (loop) とも呼びます。

[6] Ruby では「`!`」は「否定」を表すのに使っています。階乗の記号ではないので注意してください。

[7] Ruby ではさらに演算子として `and`、`or`、`not` も使えますが、結合の強さが記号版と違っていて混乱しやすいので、本資料では使っていません。

かつ (両方が成立)	または (最低限一方が成立)	否定 (〜でない)
条件1 && 条件2	条件1 || 条件2	! 条件1

では例として、「入力 x の絶対値を計算する」ことを考えます。擬似コードを示しましょう。

- abs1: 数値 x の絶対値を返す
- 　もし $x < 0$ ならば、
- 　　$result \leftarrow -x$。
- 　そうでなければ、
- 　　$result \leftarrow x$。
- 　枝分かれ終わり。
- 　$result$ を返す。

考え方としては簡単ですね? これを Ruby にしてみましょう。

```
def abs1(x)
  if x < 0
    result = -x
  else
    result = x
  end
  return result
end
```

実行の様子も示しておきます (0 もテストしていることに注意。作成したコードをテストするときには系統的に洩れなく試してみることが大切です)。

```
irb> abs1 8        ←正の数の絶対値は
=> 8               ←元のまま
irb> abs1 -3       ←負の数であれば
=> 3               ←正の数になる
irb> abs1 0        ← 0 の場合も
=> 0               ←元のまま
irb>
```

ところで、同じ絶対値のプログラムを次のように書いたらどうでしょうか?

- abs2: 数値 x の絶対値を返す
- 　もし $x < 0$ ならば、
- 　　$-x$ を返す。
- 　そうでなければ、
- 　　x を返す。
- 　枝分かれ終わり。

Ruby 版は次のようになります。

```
def abs2(x)
  if x < 0
    return -x
  else
    return x
  end
end
```

先のとどちらが好みでしょうか? また、別のバージョンとして次のものはどうでしょうか?

- abs3: 数値 x の絶対値を返す
- 　$result \leftarrow x$。
- 　もし $x < 0$ ならば、
- 　　$result \leftarrow -x$。
- 　枝分かれ終わり。
- 　$result$ を返す。

Ruby プログラムも示します (`if` を 1 行に書いてみました。このような時は `then` が必須)。

```
def abs3(x)
  result = x
  if x < 0 then result = -x end
  return result
end
```

3 つのプログラムについて、あなたはどれが好みだったでしょうか?

一般に、プログラムの書き方は「どれが絶対正解」ということはなく、場面ごとに何がよいかが違ってきますし、人によっても基準が違うところがあります。ですから、皆様がこれからプログラミングを学習するに当たっては、自分なりの「よいと思う書き方」を発見していく、という側面が大いにあります。そのことを心に留めておいてください。

演習 1 絶対値計算プログラムの好きなバージョンを打ち込んで動かせ。動いたら、枝分かれを用いて、次の動作をする Ruby プログラムを作成せよ。

a. 2 つの異なる実数 a、b を受け取り、より大きいほうを返す。

b. 3 つの異なる実数 a、b、c を受け取り、最大のものを返す。(やる気があったら 4 つでやってみてもよいでしょう。)

c. 実数を 1 つ受け取り、それが正なら「`'positive'`」、負なら「`'negative'`」、零なら「`'zero'`」という文字列を返す。

3.3 繰り返しと while 文

ここまででは、プログラム上に書かれた命令はせいぜい 1 回実行されるだけでしたから、プログラムが行う計算の量はプログラムの長さ程度しかありませんでした。しかし、繰り返しがあれば、その範囲内の命令は何回も反復して実行されますから、短いプログラムでも大量の計算を行わせられます。

まず、繰り返しの最も一般的な形である、条件を指定した繰り返しの擬似コードは次のように書き表すものとします。[8]

- ～ である間繰り返し、
- 　動作 1。
- 繰り返し終わり。

この形の繰り返しは、Ruby では **while 文** (while statement) として記述します。

```
while 条件 do
  ... 動作 1 ...
end
```

[8]「～」のところには条件を記述しますが、ここに書けるものは if 文の条件とまったく同じです。

条件の次にある do も、Ruby では省略することができます。ただし、「動作 1」を条件と同じ行に書く場合は省略できません。本資料では do は省略しないことにします。

多くのプログラミング言語では、このような条件を指定した繰り返しは while というキーワードを用いて表すので、**while ループ**と呼びます。while ループは形だけなら if 文より簡単ですが、慣れるまではどのように実行されるかイメージが湧かない人が多いと思います。while ループの実行のされ方は、次のようなものだと考えてください。

- 「〜」を調べる (成立)。
- 動作 1 を実行。
- 「〜」を調べる (成立)。
- 動作 1 を実行。
- 「〜」を調べる (成立)。
- 動作 1 を実行。
- …
- 「〜」を調べる (不成立)。
- 繰り返しを終わる。

つまり、条件を調べ、成り立てば動作 1 を実行し、また条件を調べ、…のように繰り返していき、条件が成り立たなくなると繰り返しを終わります。

3.4 例題: 平方根の計算

while を使った例として、平方根を求める計算をやってみましょう (`Math.sqrt(`x`)` で平方根が求められるということになっていましたが、その計算を自前でやるとどうかということです)。ここでは、次のような考え方でやってみます (簡単のため $x > 1$ とします)。

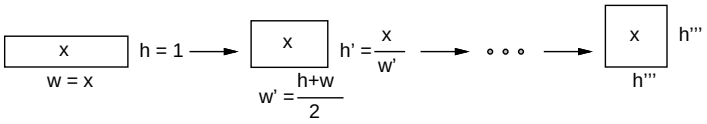

図 2: 平方根を求める

(1) 高さ h が 1、面積が x の長方形を考えます。幅 w は当然 x ですね (図 2 左)。

(2) 高さと幅の平均を計算し、幅 w をこの値に変更します。ここで面積は x のままにしたいので、高さ h は $\frac{x}{w}$ に変更します (図 2 中)。

(3) 上記の (2) を何回も繰り返すと、h と w はどんどん近付いて行き、ほとんど等しくなります。そのときは、幅も高さもほぼ同じなので、これを h とすると $h \times h = x$ となります。つまり h は x の平方根です (図 2 右)。

なんだかずるい、と思いましたか? いくら近くても本当の平方根ではない? しかし思い出してください。コンピュータによる実数計算はどのみち近似値ですし、2 の平方根などは無理数つまり小数点以下が無限に続く数なので、どのみち「本当の値」は表現できません。ですから割り切って、誤差 ϵ を指定し、その精度で平方根を計算する、ということにします。擬似コードは次のようになります。

- sqrt1: 数値 x の平方根を誤差 e で求めて返す
- 　$w \leftarrow x$。$h \leftarrow 1$。
- 　$w - h > e$ である間繰り返し、
- 　　$w \leftarrow \frac{w+h}{2}$。$h \leftarrow \frac{x}{w}$。
- 　繰り返し終わり。

- h を返す。

これを Ruby に直したものを示します。なお、Ruby では 1 行に複数の文を書くときは「;」で区切る必要があります。

```
def sqrt1(x, e)
  w = x; h = 1.0
  while w - h > e do
    w = 0.5 * (w + h); h = x / w
  end
  return h
end
```

実行してみましょう。

```
irb> sqrt1 2, 0.0001
=> 1.41421143847487
irb> sqrt1 2, 0.0000001
=> 1.4142135623715002
```

確かに、誤差を小さく指定するとその分だけ精度が高くなります。

3.5 計数ループ

ところで、上のプログラムでは誤差を指定して「差が誤差を下回るまで」という条件で繰り返していましたが、実はこの計算方法だと 10 回も繰り返せば十分な精度が得られます。では「10 回」繰り返すにはどうしたらいいでしょう。一般に回数を `n` とすると、次のようなループを書けばよいのです。(**カウンタ** (counter) とは「数を数える」ために使う変数のことを言います)。

```
i = 0          # i はカウンタ
while i < n do # 「n 未満の間」繰り返し
  ...          #  ここでループ内側の動作
  i = i + 1    # カウンタを 1 増やす
end
```

このように指定した上限まで数えながら反復する繰り返しを**計数ループ** (counting loop) と呼びます。計数ループはプログラムで頻繁に使われるため、ほとんどのプログラミング言語は計数ループのための専用の機能や構文を持っています (while 文でも計数ループは書けますが、専用の構文のほうが書きやすくて読みやすいからです)。

Ruby では計数ループ用の構文として **for 文** (for statement) を用意しています。これを使って上の while 文による計数ループと同等のものを書くと次のようになります。

```
for i in 0..n-1 do
  ...
end
```

これは、カウンタ変数 `i` を 0 から初めて 1 つずつ増やしながら `n-1` まで繰り返していくループとなります (多くのプログラミング言語では、計数ループを表すのに for というキーワードを使うので、計数ループのことを **for ループ**と呼ぶこともあります)。

せっかく for 文を説明しておきながら恐縮ですが、以下では計数ループを整数値が持つメソッド `times` を使って書くことにします。[9] これはたとえば次のようになります。

[9] なぜ for 文でなく `times` を使うかというと、ブロックを受け取るメソッドは Ruby でさまざまな用途に使える便利な仕組みなので、そちらに慣れたほうがよいと思うからです。

```
10.times do
  ...
end
```

このtimesというのは、整数値 (上の場合は 10 です) に「対して呼び出せる」メソッドであり、さらにブロック (コードの並び、do〜endの部分) を受け取るようになっています。そしてそのブロックを数値の回数 (上の例では 10 回) 実行します。ブロックの指定のための do は省略できません。[10] ではこれを使って、平方根のプログラムを書き直してみました。回数を決めたので e は渡す必要がありません。

```
def sqrt2(x)
  w = x; h = 1.0
  10.times do
    w = 0.5 * (w + h); h = x / w
  end
  return h
end
```

ところで、計数ループの中でカウンタの値 $(0, 1, 2, \ldots)$ を使いたいこともあります。このため、timesは各繰り返しごとにカウンタ値をブロックにパラメタとして渡してくれます。上の例ではそれを受け取っていませんでしたが、ブロックの冒頭に「|名前|」という書き方でパラメタ (の列) を指定することで、このパラメタを受け取ることができます。先の例題を少し直して、値の変化を見てみましょう。このとき「何回目」かも表示させるのに times のパラメタを受け取ります。

```
def sqrt2x(x)
  w = x; h = 1.0
  10.times do |i|
    w = 0.5 * (w + h); h = x / w
    puts "#{i}: w = #{w}, h = #{h}"
  end
  return h
end
```

このプログラムでは途中の表示に puts と文字列の中への式の埋め込みを使っています。実行のようすを見ましょう。

```
irb> sqrt2x 2
0: w = 1.5, h = 1.3333333333333333
1: w = 1.4166666666666665, h = 1.411764705882353
2: w = 1.4142156862745097, h = 1.41421143847487
3: w = 1.4142135623746899, h = 1.4142135623715002
4: w = 1.414213562373095, h = 1.4142135623730951
5: w = 1.414213562373095, h = 1.4142135623730951
6: w = 1.414213562373095, h = 1.4142135623730951
7: w = 1.414213562373095, h = 1.4142135623730951
8: w = 1.414213562373095, h = 1.4142135623730951
9: w = 1.414213562373095, h = 1.4142135623730951
=> 1.4142135623730951
```

[10] それで混乱しやすいので、while でも do を省略しないことにしたわけです。

確かに、5 回目から先はもう値は変化していないことが分かります (カウンタは 0 から始まることに注意)。

演習 2 上の演習問題のプログラムを打ち込んで動かせ。動いたら「誤差を指定する版」でも途中経過と回数が表示させるように直して動かしてみよ。

演習 3 次のような、繰り返しを使ったプログラムを作成せよ。

a. 整数 n を受け取り、2^n を計算する。
b. 整数 n を受け取り、$n! = n \times (n-1) \times \cdots \times 2 \times 1$ を計算する。
c. 整数 n と整数 $r(\leq n)$ を受け取り、${}_nC_r$ を計算する。

$$ {}_nC_r = \frac{n \times (n-1) \times \cdots \times (n-r+1)}{r \times (r-1) \times \cdots \times 1} $$

4 制御構造の組み合わせ

少し込み入ったプログラムになると、ある制御構造 (枝分かれ、繰り返し) の内側にさらに制御構造を入れることになります。たとえば、

- もし〜であれば、
- 　条件〜が成り立つ間繰り返し、
- 　　○○をする
- 　以上を繰り返し。
- 枝分かれ終わり。

だと次のようになるわけです。

```
if ...
    while
        ...
    end
end
```

このように規則に従って要素を組み合わせて行くことで (単に並べるのも組み合わせ方のうち)、いくらでも複雑なプログラムが作成できます。これはちょうど、簡単な規則と単語からいくらでも複雑な文章が (日本語や英語で) 作れるのと同じです。

演習 4 a と b の最大公約数を $gcd(a,b)$ と記す。正の整数 x、y の $gcd(x,y)$ を求めることを考える。

- $x = y$ のとき、$gcd(x,y) = x = y$。
- $x > y$ のとき、$gcd(x,y) = gcd(x-y,y)$。
- $x < y$ のとき、$gcd(x,y) = gcd(x,y-x)$。

これを利用して、2 つの正の整数 x、y に対してその最大公約数を求めるアルゴリズムの疑似コードを書き、Ruby プログラムを作成せよ (なぜこれで求まるかも説明すること)。

演習 5 「正の整数 N を受け取り、N が素数なら `true`、そうでなければ `false` を返すれる Ruby プログラム」を書け。[11] まず疑似コードを書き、次に Ruby に直すこと。(ヒント: N が素数ということは、N を 2〜$N-1$ のいずれで割っても余りが出るということ。剰余は演算子「`%`」で計算できる)。

演習 6 「正の整数 N を受け取り、N 以下の素数をすべて打ち出す Ruby プログラム」を書け。待ち時間 10 秒以内でいくつの N まで処理できるか調べて報告せよ。(もちろん N が大きくなるように工夫してくれるとなおよい。)

[11] `true`/`false` は「はい/いいえ」を表す値である。

本日の課題 2A

「演習 1」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. プログラムを打ち込んで動かすのに慣れましたか?

Q2. 自分にとって次の「難しいポイント」は何だと思いますか?

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 2B

「演習 1〜演習 6」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。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. 枝分かれや繰り返しの動き方が納得できましたか?

Q2. 枝分かれと繰り返しのどっちが難しいですか? それはなぜ?

Q3. 課題に対する感想と今後の要望をお書きください。