情報科学 2007　久野クラス　＃2

久野　靖*

2007.10.19

はじめに

1 週間ぶりのごぶさたでしたが、演習とかやってみていかがでしたか? アンケートで色々感想を頂いていますが、まあそれなりという感じのようですので、今後とも同様に行きたいと思います。本日の目標は次の通り:

- 基本的な制御構造 (枝分かれ、繰り返し) について理解し、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/プログラムが考えられるようになる。

1　前回の演習問題の解答例 (一部)

1.1　演習 3a

まず演習 3a だが、元の例題より一層簡単ですね?

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

動かしているところ。

```
irb(main):020:0> add(3.5, 6.8)
=> 10.3
```

四則つまり和、差、商、積の場合も上とおんなじようにやればいいが、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はないだろうか (半分くらいは前回やっていない内容の紹介を兼ねています)。

まず、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る。

```
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。

```
irb(main):022:0> shisoku0(3.3, 4.7)
8.0
-1.4
15.51
0.702127659574468
=> nil
```

*筑波大学大学院経営システム科学専攻

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) が返されている。

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知れない。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よう。Ruby では文字列 (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すが、ダブルクォートの方は内部に色々なものを埋め込む機能がついている (そういうことがしたくない場合はシングルクォートを使えばよい)。具体的には、文字列の中に「`#{...}`」という形のものがあると、中かっこ内の式を評価 (値を計算すること) して、結果をそこに埋め込んでくれる。これを利用した「四則演算」のメソッドを示す。

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

実行しているところ。

```
irb(main):024:0> shisoku1(3.3, 4.7)
=> "8.0 -1.4 15.51 0.702127659574468"
```

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

もう 1 つ、Ruby など多くの言語では値の並んだものを「配列」という機能で扱う。Ruby では配列は `[...]` の中に値が並んだものとして書けるので、この形で 4 つの数値を並べることもできる。

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

実行しているところ。

```
irb(main):025:0> shisoku2(3.3, 4.7)
=> [8.0, -1.4, 15.51, 0.702127659574468]
```

似たようなものだけど、今度は配列として表示されている。こちらの方が、返された値から「0 番目」「1 番目」など番号を指定して取り出せるので便利かも知れない。配列については、次回もっと詳しく説明する。

1.2 演習 3b

これはやってくれた人が多そうだが、「`%`」で試すだけ。

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

実行してみよう。

```
irb(main):004:0> jouyo(8, 5)
=> 3
irb(main):005:0> jouyo(20, 5)
=> 0
irb(main):006:0> jouyo(-8, 5)
=> 2
irb(main):007:0> jouyo(-21, 5)
=> 4
```

ちゃんとマイナスのときも試しましたか? ここで「マイナスだとどうだろう」とか思うようになって欲しいわけです。で、マイナスのとき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、という風に考えるのでしょうね。じゃあ割る数がマイナスだったら?

```
irb(main):008:0> jouyo(8, -5)
=> -2
irb(main):009:0> jouyo(-8, -5)
=> -3
```

今度は 2 数ともマイナスの場合をまず考えて、それから等間隔で、というふうに考えるといいのでしょうね。まあ、こういう風にするのが何かと便利なのだろうと思います。

1.3 演習 3c+3d

演習 3c と 3d は円錐の体積と表面積…表面積はけっこう面倒ですよね。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 で割るだけ。表面積は展開図を考えると、底面の面積はもう分かってるから側面は…側面の半径は $\sqrt{r^2+h^2}$。で、これを l と置くと、側面になる扇型の中心角は、全円周 (ラジアンで表した中心角 2π の場合に相当) の長さ $2\pi l$ と底面の円周の長さ $2\pi r$ の比率 $\frac{r}{l}$ に 2π を掛けたもの。で、扇型の面積は半径 l の円の面積の $\frac{r}{l}$ 倍となるわけだ。以上から表面積は

$$\pi r^2 + \pi l^2 \frac{r}{l} = \pi(r^2 + rl)$$

こういうのが得意な人むけにちょっと遊ぼうと思ったのだけど…大変でしたか?

```
def cornvol(r, h)
  return (r*r*3.1416*h) / 3.0
end
def cornarea(r, h)
  l = Math.sqrt(r**2 + h**2)
  return 3.1416 * (r**2 + r*l)
end
```

ちなみに「**」はべき乗演算子。説明してないことが次々に出て来てすみません。もちろん「r*r」とかでも書けるわけなので。実行例。

```
irb(main):029:0> cornvol(3.0, 4.0)
=> 37.6992
irb(main):030:0> cornarea(3.0, 4.0)
=> 75.3984
```

「円周率は 3.1416 じゃないねん!」と思う人もいたようだが…計算機での計算は「電卓での計算」と同じであり、有限の桁数でしか行なえないのであり、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。もっとも、3.141592653589793 くらいまでは扱えるので、それをいちいち書くのは嫌だという人のために `Math::PI` と書いてもいい (ついでに e は `Math::E`)。

ところで、ここまでに出て来たものが自分が作ったのと違う、と思いました? もちろん、新しい機能の予告に習ってないことを使って見せたというのはあるのですが、それは別としても 1 つのことをするプログラムが複数あるのは当然であり、題意に会ってさえいれば、どっちが 1 つだけが正解ということはない。ちょうど日本語で同じことを言うのにさまざまな言い方があるのと同じこと。ただし、スマートだとか短いとか分かりやすいとかそういった点で違いはあるかもしれない。美観の問題! だから自分の好みもとりまぜて、考えて選ぶこと。

あと、字下げ (左側に空白を入れること) をしない人がいるけど、これは絶対やめた方がよい。たとえば、

```
def ....      def ...
  ....        ....
  ....        ....
end           end
```

左のように書いてあれば、どこまでがメソッドの範囲かすぐ分かる。しかし右のようにべったり揃えてしまうとそれが分からないし、たとえば間違って end をつけ忘れたり別の場所に入れてしまうともう大混乱になる。たかがスペースキーを打つ手間を惜しんで後で間違い探しに 1 時間も掛けるのは阿呆みたいでしょ? プログラムは「美しく」書こう。

1.4 演習 3e+3f

演習 3 の残りは省略するが、要は色々やってみて (もちろん何をやってみようかきちんと考えることは必要)、その結果を報告すればいいのではないでしょうか。

1.5 演習 4

時間の計測方法については前回資料に載っていたのでいいですね。さっそくあれでやってみよう。

```
% cat r1ex4.rb
def add(x, y)
  return x + y
end

def bench(count, &block)
  t1 = Process.times.utime
  count.times do yield end
  t2 = Process.times.utime
  return t2 - t1
end
% irb
irb(main):001:0> load 'r1ex4.rb'
=> true
irb(main):002:0> bench 100000 do add(1, 1) end
=> 0.125
irb(main):003:0> bench 100000 do add(10000, 1) end
=> 0.125
irb(main):004:0> bench 100000 do add(100000000, 1) end
=> 0.125
irb(main):005:0> bench 100000 do add(1000000000000, 1) end
=> 0.21875
```

ほうほう、そうすると、1 億と 1 京 (1 億× 1 万) の間に「ビット数を増やすところ」があるようだ。それはたぶん、2^N の数値付近なはず。32 ビットの整数だと、正の数は $2^{31}-1$ まで表せるので、この値で試してみましょう。この値は…電卓は使わなくても、Ruby で計算すればよい。それにはプログラムを書く必要はなくて、irb で式を実行すればいい。

```
irb(main):006:0> 2**31 - 1        ←「**」はべき乗演算子
=> 2147483647
irb(main):007:0> bench 100000 do add(2147483647, 1) end
=> 0.2265625
irb(main):008:0> bench 100000 do add(2147483646, 1) end
=> 0.21875
irb(main):009:0> bench 100000 do add(2147483645, 1) end
=> 0.2265625
```

うーん、変わらない…ということは、多倍長になる「切れ目」はもっと小さいところにあるらしい。では $2^{30}-1$ ではどうだろう。

```
irb(main):010:0> 2**30 - 1
=> 1073741823
irb(main):011:0> bench 100000 do add(1073741823, 1) end
=> 0.265625
irb(main):014:0> bench 100000 do add(1073741822, 1) end
=> 0.125
irb(main):013:0> bench 100000 do add(1073741821, 1) end
=> 0.125
```

ビンゴでした。ところで、じゃあ「次の」大きくなる境目はどうか…やってみると分かるが、今度は上記ほとはっきりと違いが分からない。つまり $2^{31}-1$ までが扱える整数の表現と、それより大きい整数の表現とは大きく違うが、大きい方は桁数に応じて処理時間がちょっとずつ長くなる程度なのだろう、ということが分かる…と思う。

ところで余談ですが、この実験では要するに足し算をすればいいので、わざわざ add を呼ばなくても次のように足し算を do-end の中に直接書いて実験してもよい (その分速くなるが、段差ができる値は変わらない)。

```
irb(main):035:0> bench 100000 do 1073741823 + 1 end
=> 0.2109375
irb(main):036:0> bench 100000 do 1073741822 + 1 end
=> 0.078125
```

1.6 演習 5

演習 5 は前回資料に書いたように printf を使うとよく分かる。いちいち関数を書かずに irb だけで直接計算してみよう。

```
irb(main):037:0> printf "%.20g\n", 1/3
0                      ←おっと! 整数同士の除算だと 0 でした
=> nil                 ←この nil は printf が値を返さないため
irb(main):038:0> printf "%.20g\n", 1.0/3.0
0.33333333333333331483 ←有効桁数は 10 進で 16 桁程度だと分かる
=> nil
irb(main):039:0> printf "%.20g\n", 1.0/3.0 * 3.0
1                      ←それに 3 を書けると最後の桁が丸められて元に戻る
=> nil
irb(main):040:0> printf "%.20g\n", 7.0 / 10.0
0.69999999999999995559 ← 0.7 は 2 進表現では切りがよくないので
=> nil
irb(main):041:0> printf "%.20g\n", 7.0 * 0.1
0.70000000000000006661 ← 0.1 という値も誤差を含むため上と微妙に違う
=> nil
```

このように、有限桁数の計算なので微妙に誤差が現れる。しかし一方で、2 進表現を使っているということは、2^N とか $\frac{1}{2^N}$ とかは非常に「切りのいい数」となり、あふれない限りは誤差がない。

```
irb(main):048:0> printf "%.40g\n", 7.0 / 16.0
0.4375
=> nil
irb(main):049:0> printf "%.40g\n", 7.0 * 0.0625
```

```
0.4375
=> nil
irb(main):050:0> printf "%.40g\n", 7.0 / (2**16)
0.0001068115234375
=> nil
irb(main):051:0> printf "%.40g\n", 7.0 * (0.5**16)
0.0001068115234375
=> nil
irb(main):052:0> printf "%.40g\n", 7.0 / (2**32)
1.62981450557708740234375e-09
=> nil
irb(main):053:0> printf "%.40g\n", 7.0 * (0.5**32)
1.62981450557708740234375e-09
=> nil
```

なお、最後に出て来る「e-09」とかは指数表記の記法で、$\times 10^{-9}$ を表しています。たとえば光の速さは 3.0e+12m/s ということになる。

演習 6 は略しますが、NaN とか∞とかが現れることが分かっていればとりあえずいいんではないでしょうか。

2 アルゴリズムと制御構造

2.1 基本的な制御構造

ここまでに出て来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、一番下まで来たらそれでおしまい、というものだった。それでも単純な計算なら問題なくできるが、手順が複雑になってくると「枝分かれ」や「繰り返し」などが必要になってくる。このような、実行の流れを表す構造のことを一般に制御構造と呼ぶ (動作を順番に書くと順番に実行される、というのも実は「連接」という制御構造ということになっている)。

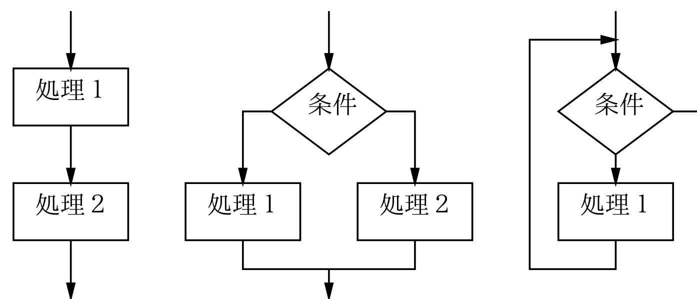

図 1: 3 つの基本的な制御構造

この 3 種類の構造 (連接、枝分かれ、繰り返し) は図 1 のような流れ図に対応している。それで、なぜこの 3 つが基本的かというと、「どんなにごちゃごちゃの流れ図でもこの 3 つの組み合わせに書き換えられる」という定理があり、そのためにこの 3 つだけあればどのような処理の流れでも表現可能だからである。

たとえば、図 2 左のような「入り組んだ」流れ図があるとする。これをこのまま「何をするか」理解するのは大変だが、同じ動作だが基本的な制御構造の組み合わせになるような流れ図に (中) に書き換え、さらに条件の組み合わせを見て簡単化する (右) と、何をするかがずっとよく分かるようになる。[1]

[1]右側の流れ図を見ると、「x > y なら x+1 を出力」はすぐわかる。繰り返しの中では、x が y より小さい間 x を 1 増やし、y を 1 減らすので、x と y が同じ値に近付くことになる。x+y が偶数ならちょうど等しくなって終わるが、奇数なら x が y より 1 多くなる。このため、「x ≤ y なら $\lfloor \frac{x+y}{2} \rfloor$ を出力」と分かる。

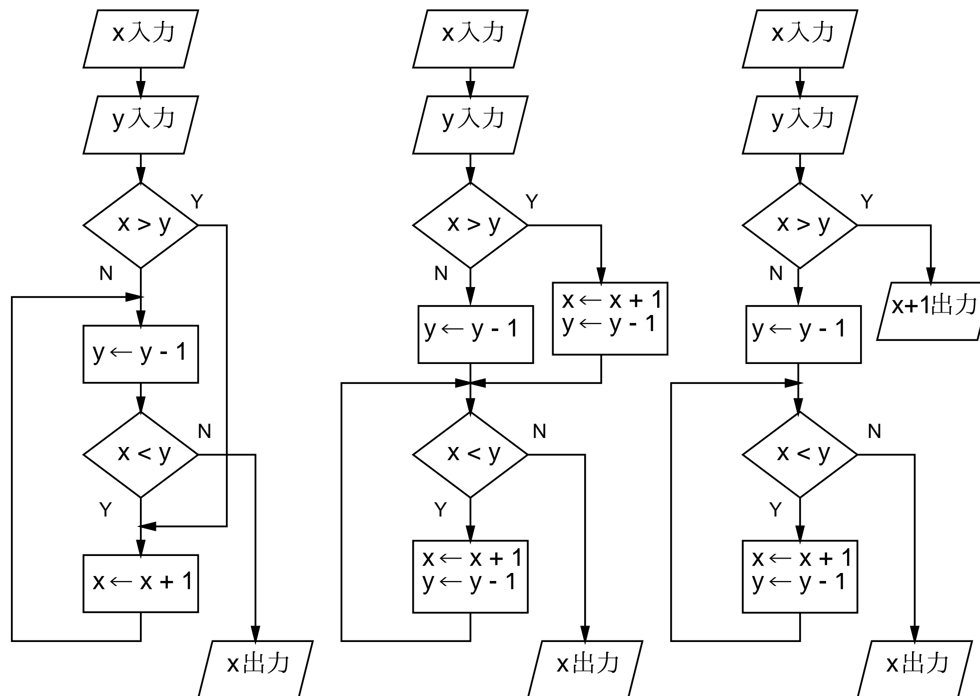

図 2: 分かりにくい流れ図と構造に従った流れ図

このため、現在ではアルゴリズムを 3 種類の構造の組み合わせで表すのがコンピュータサイエンス分野では標準的となってきていて、そのため「入り組んだ」構造が簡単に書けてしまう流れ図の代わりに疑似コードが使われるようになっている。

2.2 枝分かれと if 文

さて、ここまでに説明した機能だけでは、上から順に動作を実行していくようなプログラムしか作れない。それでは不便なので、枝分かれを学ぶ。枝分かれの疑似コードは次のような形で書く:

- 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。

なお、「動作 2」が不要なら「そうでなければ」も書かなくてよい。これを Ruby では次のように「if 文」と呼ばれる文を使って書く (右は「動作 2」のない場合):

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

`then` は Ruby では書いても書かなくてもよい (読みやすさのために書いてもいいようになっている。ただし、「動作」を条件と同じ行に書く場合は `then` を省略できない)。「条件」については、当面は次のものがあると思っておいて欲しい。

- 比較演算 — たとえば「`x > 10`」のようなもの。`>`(より大)、`>=`(以上)、`<`(より小)、`<=`(以下)、`==`(等しい)、`!=`(等しくない) がある。[2]

[2]Ruby では「`!`」は「否定」を表すのに使っている。階乗ではないので注意。

- 条件の組み合わせ — 「条件 1 `&&` 条件 2」(A かつ B)、「条件 1 `||` 条件 2」(A または B)、「`!`条件 1」(A でない) が使える。ここまでは C や Java と同じだが、Ruby ではさらに `and`、`or`、`not` も使える。[3]そして、複数のかつ、またはを組み合わせることもでき、適宜かっこを使ってもよい。

2.3 例題: 絶対値の計算

では具体的な例題として、「入力 x の絶対値を計算する」ことを考えよう。まず疑似コードを示す:

- 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 にしてみよう:

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

ところで、同じプログラムだがこう書いたらどうだろう?

- 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 版は次の通り。

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

先のとどちらが好みだろうか? また、別のバージョンとして次のものはどうだろう?

- abs3: 数値 x の絶対値を返す
- 　数値 x を入力する。
- 　$abs \leftarrow x$。
- 　もし $x < 0$ ならば、

[3] Ruby では `and`、`or`、`not` の方が記号の演算子よりも結び付きの強さが弱くなっている。たとえば「`A || B and C || D`」は全部記号バージョンで書くと「`(A || B) && (C || D)`」となる。

- 　　　$abs \leftarrow -x$。
- 　枝分かれ終わり。
- 　数値 x とその絶対値 abs を出力する。

このように、「そうでなければ」の部分で何もすることがなければ「そうでなければ」以下を書かないでよい。Ruby プログラムを示す。

```
def abs2(x)
  result = x
  if x < 0 then result = -x end
  return result
end
```

`if` を 1 行に書いてみたが、Ruby でもこのような時は `then` が必要。それで、3 つのプログラム、どれが好みですか?

要するに、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くるし、人によっても基準が違うところがある。早く自分の基準を見つけよう!

演習 1 上の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そのまま動かせ。

演習 2 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、より大きい方を返す。

b. 3 つの異なる実数 a、b、c を受け取り、最も大きいものを返す。やる気があったら 4 つでやってみてもよい。

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。

2.4 繰り返しと while 文

枝分かれができても、まだプログラムで大したことができる気がしない。計算機の特徴は、どんなつまらない単純作業でもいくらでも繰り返してくれることにあるので、繰り返しが使えるとプログラムもぐっと役に立つ。

一般の条件に基づく繰り返しを疑似コードで記述するときは、次のように書く:

- ～ である間繰り返し、
- 　動作 1。
- 繰り返し終わり。

「～」のところには条件を記述する。書けるものは if 文の条件とまったく同じである。この繰り返しを Ruby で書く場合は「while 文」を使う:

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

この do も Ruby では省略してもよい (読みやすさのために書いてもよい)。

while 文は形だけなら if 文より簡単だが、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かも知れない。たとえば次のような感じで考えれば分かりやすいだろう:

- 「～」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そこで終わるわけである。

2.5 例題: 数値積分

繰り返しの具体的な題材として**数値積分**を取り上げて見よう。定積分を求めるのって、積分の公式を覚えたりあてはめ変形に苦労したり、大変でしたよね? もうそんな心配はない、コンピュータがあれば元の関数から直接計算してしまえる (実はそんなにいいものでもないけど)。

関数 $y = f(x)$ の $x = a$ から $x = b$ までの定積分というのは図 3 のように、その $[a, b]$ 部分の関数の下側の面積、ですよね? (なぜだかは聞かないように、私は理系だけど数学は大嫌いです。)

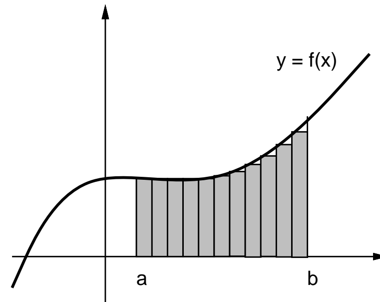

図 3: 数値積分の原理

だったら、この図 3 にあるように、その部分に多数の細長い長方形を詰めてみて、その面積を合計すればいいわけだ。幅は区間を n 当分した値 dx にするものとして、高さは $f(x)$ を計算すれば済むので簡単である。この方法だったら $f(x)$ が解析的に不定積分が求められないヘンテコな関数だろうと計算するだけだからへっちゃらである。

とはいえ、今はプログラムを作って正しい値が求まるかどうかチェックしたいので、ひじょーに簡単な関数 $y = x^2$ でやってみる。不定積分は $\frac{1}{3}x^3$ だから、区間 $[a, b]$ の定積分は $[\frac{1}{3}x^3]_a^b$ ということになる。たとえば $[1, 10]$ だったら $\frac{1000}{3} - \frac{1}{3} = \frac{999}{3} = 333$ ということになる。

ではアルゴリズムを作って見る。

- integ1: 関数 x^2 の区間 $[a, b]$ の定積分を区間数 n で計算
- 　　$dx \leftarrow \frac{b-a}{n}$。
- 　　$s \leftarrow 0$。
- 　　$x \leftarrow a$。
- 　　$x < b$ が成り立つ間繰り返し:
- 　　　　$y \leftarrow x^2$。 # 関数 $f(x)$ の計算
- 　　　　$s \leftarrow s + y \times dx$。
- 　　　　$x \leftarrow x + dx$。
- 　　繰り返しおわり。
- 　　s を返す。

要するに、x にまず a を格納しておき、繰り返しの中で $x \leftarrow x + dx$、つまり x に dx を足した値を作ってそれを x に**入れ直す**ことで x を徐々に (dx きざみで) 動かして行き、b まで来たら繰り返しを終わる。このように、繰り返しは「こういう条件で変数を動かして行きこうなったら終わる」という考え方が必要なわけである。面積の方は、s を最初 0 にしておき、繰り返しの中で細長い長方形の面積を繰り返し加えて行くことで、合計が求まるわけだ。

なお、「#」の後ろに書いたのは注記 (コメント) で、Ruby 言語でもこの書き方でプログラム中に覚え書きを入れておくことができる。では Ruby プログラムを示そう。

```
  def integ1(a, b, n)
    dx = (b - a) / n;
    s = 0.0
    x = a
#   count = 0
    while x < b
```

```
      y = x**2         # 関数 f(x) の計算
      s = s + y * dx
      x = x + dx
#     count = count + 1
#     puts "count=#{count} x=#{x}"
    end
    return s
  end
```

3 箇所ほど行頭に「#」があって行全体がコメントになっているが、これは後で使う。とりあえずないものと思って読んで欲しいが、やっていることは上の疑似コードそのままである。さて、333 が求まるだろうか? 実行させてみる:

```
irb(main):023:0> load 'sam22.rb'
=> true
irb(main):024:0> integ1(1.0,10.0,100)
=> 337.557149999999            ←ふーん?
irb(main):025:0> integ1(1.0,10.0,1000)
=> 332.554621500007            ←小さい
irb(main):026:0> integ1(1.0,10.0,10000)
=> 333.045451214912            ←大きい…
```

なんだかヘンである。そこで、繰り返しの回数がいくつになっているかをチェックすることにして、上の行頭の「#」を削って (変数 count に回数を数えつつ x を表示するようにして) 動かし直してみた:

```
irb(main):002:0> integ1(1.0,10.0,100)
...
count=98 x=9.81999999999999  ←誤差が…
count=99 x=9.90999999999999
count=100 x=9.99999999999999
count=101 x=10.09            ← 101 回目が…
=> 337.557149999999          ←このため多い
```

区間を 100 個でやってみると、長方形を 1 個余計に加えてしまってそれで多すぎるようだ。しかしなぜこんなことが起きるのだろうか? それは、$x \leftarrow x + dx$ で x を増やしていって b になったらやめる、というアルゴリズムの問題である。そもそもコンピュータでの計算は近似値なので、最初に dx を区間長の $\frac{1}{100}$ に計算したとしても、そこには誤差がある。誤差のため、100 回足した後でもわずかに b より小さい場合には、さらにもう 1 回繰り返しを実行してしまうわけだ。

2.6 計数ループ

ではどうすればいいのだろうか。繰り返しが 100 回と決めているのだから、回数を数えるのは整数型の変数で行い (整数であればあふれない限り誤差がない)、それをもとに各回の x を計算するのがよい。つまり、次のようなループを書くことになる:

```
i = 0          # i は「カウンタ」(数を数える変数)
while i < n    # 「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
```

このようなループを「計数ループ」と呼ぶ。計数ループはプログラムで多く使うため、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る。つまり、上のような while でも計数ループは書けるのだが、専用の機能の方が書きやすく読みやすいということ。

Ruby では回数指定の計数ループは、整数値が持つメソッド `times` を使って書くのが定石である。これはたとえば次のようにする。

```
100.times do
  ...
end
```

メソッド `times` はブロック (コードのかたまり) を引数として受け取り、そのブロックを指定回数 (上の例では 100 回) 実行する。前回課題で時間計測に使ったのはこの機能だったわけだ。ブロックの範囲は `do` … `end` で指定される。このため、こちらの場合は `while` と違って `do` が省略できない。

ところで、ループの中でカウンタの値 (0、1、2、…) を使いたいことがありそうですね? 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る。上の例ではそれを捨てていたが、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れる。たとえば次のような感じである:

```
100.times do |i|
  puts i
end
```

以下では、疑似コードでもこのような計数ループを次のように書く:

- 変数 i を 0 から n の手前まで変えながら繰り返し、
-     ... `#` ループ内の動作
- 繰り返しおわり。

2.7 例題: 数値積分 (つづき)

余談が終わったので、では先の積分プログラムを計数ループを使うように直してみる:

- integ1: 関数 x^2 の区間 $[a, b]$ の定積分を区間数 n で計算
-     $dx \leftarrow \frac{b-a}{n}$。
-     $s \leftarrow 0$。
-     変数 i を 0 から n の手前まで変えながら繰り返し、
-         $x \leftarrow a + \frac{i}{n}(b-a)$。
-         $y \leftarrow x^2$。 `#` 関数 $f(x)$ の計算
-         $s \leftarrow s + y \times dx$。
-     繰り返しおわり。
-     s を返す。

先のプログラムと違うのは、毎回 x を i から計算しているところである。では、これを Rby にしたものを示す:

```
def integ2(a, b, n)
  dx = (b - a) / n;
  s = 0.0
  n.times do |i|
    x = a + i * (b - a) / n;
    y = x**2        # 関数 f(x) の計算
    s = s + y * dx
  end
  return s
end
```

これを動かしてみる:

```
irb(main):009:0> integ2 1.0, 10.0, 100
=> 328.55715
irb(main):010:0> integ2 1.0, 10.0, 1000
=> 332.5546215
irb(main):011:0> integ2 1.0, 10.0, 10000
=> 332.955451215
```

今度はきざみを小さくすると順当に誤差が減少して行くことが分かる。しかし、常に正しい面積である 333 より小さいのはなぜだろう?

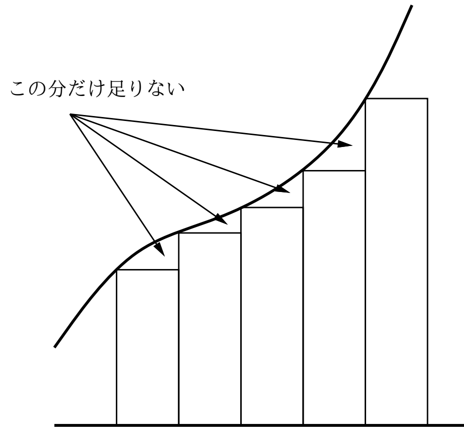

図 4: 区間の左端を使う場合の誤差

それは、長方形の面積を計算するのに微小区間の左端の x を使って高さを求めているため、増大する関数では図 4 のように微小な三角形のぶんだけ面積が小さめに計算されてしまうからである (逆に減少する関数だと大きめに計算される)。これをもうちょっと何とかする方法については、演習問題にしたのでやってみて欲しい。

演習 3 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「減少する関数だと値が大き目に出る」ことも確認せよ。できれば、左端ではなく右端で計算するのもやってみるとよい。その後、次のような考え方で誤差が減少できるかどうか、実際にプログラムを書いて試してみよ。

a. 左端の x だけでも右端の x だけでも弱点があるので、両方で計算して平均を取る。
b. 左端や右端だからよくないので、区間の中央の x を使う。
c. 上記 a と b をうまく組み合わせてみる。

演習 4 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 の階乗 $(n \times (n-1) \times \cdots \times 2 \times 1)$ 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。

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

d. x と計算する項の数 n を与えて、次のテイラー展開を計算せよ。

$$ sinx = \frac{1}{1!}x^1 - \frac{1}{3!}x^3 + \frac{1}{5!}x^5 - \frac{1}{7!}x^7 + \cdots $$

$$ cosx = \frac{1}{0!}x^0 - \frac{1}{2!}x^2 + \frac{1}{4!}x^4 - \frac{1}{6!}x^6 + \cdots $$

実際に値の分かる x を入れて精度を確認してみること。± 100 π とかだとどうか? n はいくつくらいが必要か?

3 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る。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ある。

```
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る。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) 作れるのと同じである。

演習 5 2 数 a、b の最大公約数を $gcd(a, b)$ と記すことにする。正の整数 x、y について $gcd(x, y)$ を求めることを考える。以下 (☆) に注意。

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

これを利用して、2 つの正の整数 x、y を読み込み、その最大公約数を出力するアルゴリズムの疑似コードを書き、それを実現した Ruby プログラムを作成せよ (☆の簡単な証明も書くこと)。

演習 6 「正の整数 N を受け取り、N が素数か否かを返すれる Ruby プログラム」を書け。まず疑似コードを書き、それから 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。たとえば「`7 % 4`」は 3。)

演習 7 「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。)

A 本日の課題 2A

今日は「演習 2」で動かしたプログラムを含む小レポートを久野まで電子メールで送ってください。具体的な内容は次の通り。

1. Subject: は「Report 2A」とする。[4]
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 (簡単でよい)。

   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。

[4] 注意! 1 バイトコード (いわゆる半角) で、大文字小文字もこの通りに、符号化なしで! プログラムで処理するので、この通りでないと間違って処理される可能性がいくらか高くなります。もちろんチェックはしますが。

B　次回までの課題 2B

次回までの課題は「演習 3」〜「演習 7」までの (小) 課題からプログラムを 2 つ以上作ること。ただし「演習 3」から選ぶのは最大 1 個とする。

レポートは授業開始 10 分前 (10:30) 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。具体的な内容は次の通り。

1. Subject: は「Report 2B」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 選んだプログラム 1 つのソース。
4. 説明と考察。
5. 選んだプログラムもう 1 つのソース。
6. 説明と考察。
7. 下記のアンケートの回答。

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。