情報科学　2007　久野クラス　＃1

久野　靖*

2007.10.12

はじめに

こんにちは、久野です。今週から「情報科学 (金曜 2 限)」を開始します。この科目の目標は「情報科学の基本概念や思考方法をプログラミングを通して習得すること」となっています。ですから、学習内容としては理論的なものが含まれますが、それを「確認する」ためにプログラミングができることが必要です。

久野の基本的な信念は、「プログラミングができるようになるためには、コードを自分で書いて動かす経験を一定量積むことがどうしても必要である」というものです。

従って本クラスでは、少なくとも前半は、毎回授業時間にやって頂く課題 (出席点に相当) と、授業時間外に (次の授業までの間に) やって頂く課題とを出します。大学の授業は「講義 1 時間につき、その 2 倍の自習を行う」ことが前提となっていることに注意。この科目の場合は毎週 1.5 時間なので、その倍の 3 時間を、次回までの課題のために使用する時間のめやすとしてください。もちろん、課題をやるのに掛かる時間は人によって増減しますので、これより多い人も少ない人もいると思います。

成績については、学期末に「情報科学」の全クラス共通のペーパテストを実施することとなっていますので、試験の点数と各回の課題提出点 (出席点含む) とを 50:50 で採用して総合成績をつける予定です。

使用するプログラミング言語は、Ruby 言語を使用します (後半の方でさまざまなプログラミング言語の紹介になったら他の言語も少しだけ扱います)。Ruby 言語とその処理系に関するさまざまな情報は

`http://www.ruby-lang.org/ja/`

にあります。自宅などの Windows 上で動かしたい場合はここの「ダウンロード」ページから Windows 用バイナリを取って来て入れるとよいでしょう。

Web

このクラスの Web サイトは

`http://lecture.ecc.u-tokyo.ac.jp/~kuno/is07/`

です。**ここに掲示等を出しますからこまめにチェックしてください。**出席/レポート課題等は久野宛のメールで提出してもらいますが、**それらのメールは原則としてここで公開**します。予め了承してください (公開されて困る内容は出席/レポート課題としては送らないこと)。なお、出席/レポートメールについては、先頭が「`@@@`」(半角のアットマーク 3 つ) で始まる行は削除してから掲載しますので、自分の名前を書く行にはこのおまじないをつけることを推奨します。たとえば次のような感じです。

*筑波大学大学院経営システム科学専攻

@@@ 氏名: 久野　靖
えーと、このレポートは… (以下略)

ただし、レポート本文は隠さないこと。@@@で隠してもこちらで削除します。レポートを見ることは互いがどういうことを考えたかを知り他人から学ぶよい機会だと私は考えています。

また、皆様からの率直なご意見ご感想を伺いたいので、相談サーバにある掲示板を積極的に活用してください。質問 (Q&A) 用と雑談用があります。ハンドルでの書き込みも構わないそうですが、荒らし的行為 (他人のハンドルで書いたり毎回ハンドルを変える等) はやめて欲しいそうです。上記のページからもリンクが張ってあります。

http://www.sodan.ecc.u-tokyo.ac.jp/cgi-bin/qbbs/view.cgi : Q&A
http://www.sodan.ecc.u-tokyo.ac.jp/cgi-bin/sbbs/view.cgi : 雑談

講義内容と予定

「情報科学」で学ぶ情報の基本概念としては科目共通で次のものが挙げられています:

- 離散数学
- データのモデル化
  - 対象物，構造，関係，状態変化，相互作用のモデル化
  - データ構造，再帰，オートマトン
- 抽象化の階層
- 離散数と連続数，誤差
- 計算の手間 チューリング機械，アルゴリズム，メタアルゴリズム

取り上げる順序としては、実習に使用する Ruby 言語での学びやすさも考慮して決めて行き、最終的にはこれらをひととおりカバーするようにします。このため具体的なスケジュールとして、今年はおおよそ次のようなロードマップでやりたいと考えています (冒頭数回ぶん、後半は検討中)。

- プログラムの基本概念と数値型 — 変数、演算、代入、数値のモデル化、数値の表現とその性質
- アルゴリズムとその記述 — アルゴリズム、制御構造、問題の性質の利用、手順の抽象化、再帰
- データ型とデータ構造 — 論理値、文字、文字列、配列、レコード、オブジェクト
- 動的/再帰的データ構造 — 可変長配列、連想配列、連結リスト、階層構造、木構造とそのアルゴリズム
- 抽象データ型 — 名前とその意味、内包と外延、スタック、キュー、デック、グラフとそのアルゴリズム
- アルゴリズム解析 — 時間計算量、空間計算量、整列のアルゴリズム、グラフアルゴリズムの解析、動的計画法

上記のは概要で、細かいところはやりながら決めていく予定です。では半年間、よろしくお願いいたします。

1　プログラムとモデル化

1.1　アナログとディジタル

コンピュータとは、非常に突き詰めて言えば、ディジタル情報を扱う装置だと言える。そして、これまでであれば「画像ならカメラ」「音ならテープレコーダ」のように種類ごとに別の装置を必要としていたのに対し、計算機は「ディジタル情報であれば何でも扱える」というところが画期的に違っている。

具体的にどう、という話の前に、アナログとディジタルについておさらいをしておこう。**アナログ量** (**連続量**) とは、連続的に変化する値を表す量をいう。長さ、重さ、時間、温度、速度、力の強さなどはすべてアナログ量である。

ディジタル量 (**離散量**) とは、とびとびに変化する値を表す量をいう。ものの個数、組み合わせや場合の数など「数えられる」量がディジタル量に相当する。

量をあらわすときの表し方にも、**アナログ表現**と**ディジタル表現**とがある。たとえばアナログ表現の時計や体重計では、針の位置が連続的に変化することで現在の時刻や体重を表す。一方、ディジタル表現の時計や体重計では、時刻や体重が数字で表される。

一般に、数字で量を表すことは、その数字の桁数で決まる最小単位 (「1 秒」「0.1Kg」など) より細かい部分は省略した「とびとびの」値を表すことになるので、すべてディジタル表現に相当する。

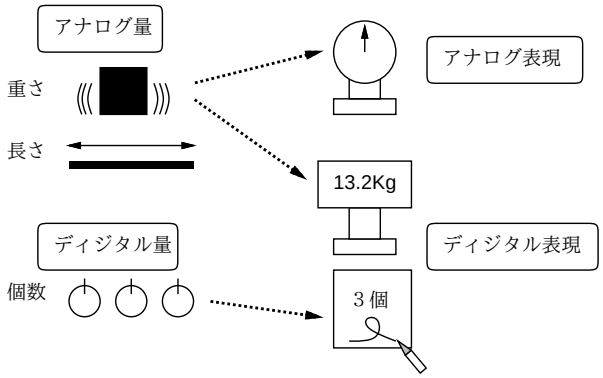

図 1: アナログとディジタル

アナログ表現は、ぱっと見ておよそどれくらいか見てとりやすいという利点があるのに対し、ディジタル表現では数字で表示させるので値を正確に読み取るのに便利だという利点がある。ただし、ディジタル表現では最小の単位よりも細かい違いは読み取れない。

また、値を記録したり伝達するのにはアナログ表現よりもディジタル表現の方が優っている。たとえば、ものの長さを記録するのに、アナログ表現であれば紐などに印をつけて覚えることになるが、紐が伸びてしまったり印がかすれてしまうなどで、後で値を正確に再現できない可能性がある。また、遠くまでその情報を伝達するのも簡単ではない。

しかし、ものさしで長さを測って数字を書き留めておけば (ディジタル記録)、数字が読めなくならない限り値を再現するのも容易であるし、数字を読み上げたりして遠くまで伝達するのも簡単である。ただし、ディジタル表現にした時点でその最小単位より細かい情報は失われていることに注意しなければならない。

以下では簡単のため、「ディジタル表現によって表されている情報」のことを単に**ディジタル情報**と呼ぶことにする。コンピュータ内部ではすべての情報はディジタル表現によって表されている。これを短く書くと「コンピュータはディジタル情報を扱う」ということになる。

1.2 コンピュータとディジタル情報

ディジタル情報とは、別の見かたをすれば「いく通りかの場合のうちのどれか」という情報であると言える。たとえば、人の体重を「少数点以下 2 桁までの Kg 単位」で表すとすると、「000.00Kg〜999.99Kg」までの 100,000 通りの場合のうちのどれか、という情報だと考えることができる (1t 以上の体重の人はいないだろうから)。

このことから、ディジタル情報の最小単位は「2 つのうちのどちらか」という情報だと考えることができる。これを「0/1 のどちらか」で表すこととし、「1 **ビット**の情報」と呼ぶ。たとえば、現在の天気を「雨が降っていない」「雨が降っている」の 2 通りの場合に分けたとすると、その情報をたとえば次のように 1 ビットの情報として表すことができる: [1][2]

ビット表現	意味
0	雨が降っていない
1	雨が降っている

1 ビットはディジタル情報の最小単位だが、複数のビットを並べた**ビット列**とすることで、より多くの情報を表現できる。たとえば、雨が降っている/いないでは大まかすぎるので、もっと詳しい情報として「晴れ」「曇」「雨」「雪」のどれであるかが知りたいとする。これは、たとえば次のように 2 ビットに対応させて表現できる。

ビット表現	意味
00	晴れ
01	曇
10	雨
11	雪

このように、ビット列の長さを 1 増やすと、表せる場合の数は 2 倍になり、一般に N ビットのビット列では 2^N 通りの場合を表すことができる。

そして、ディジタル情報とは「いく通りかの場合のうちのどれか」という情報なので、すべてのディジタル情報は (必要なだけの長さを決めることによって) ビット列で表すことができる。

コンピュータとはひらたくいえば、ビット列を蓄積/転送/加工するための装置であり、その機能によってあらゆるディジタル情報を取り扱うことができる。さらに、これから実際に見ていくように、人間の介在なしに自動的に処理を行える、という点も重要である。

1.3 モデル化とコンピュータ

モデル (model) とは、何らかの扱いたい対象があって、その対象全体をそのまま扱うのが難しい場合に、その特定の側面 (扱いたい側面) だけを取り出したものを言う。

たとえば、プラモデルであれば飛行機や自動車などの「大きさ」「重さ」「機能」などは捨ててしまい、縮尺/縮小して「形」「色」だけを取り出したもの、と言えるだろう。ファッションモデルであれば、様々な人が服を着る、その「様々さ」を捨てて特定の場面で服を見せる、という仕事だと言える (もちろんそこには服をよく見せるという意図はあるが)。

コンピュータで計算をするのに何でモデルの話をしているのだろう? それは、コンピュータによる計算自体がある意味で「モデル」だからである。たとえば、「三角形の面積を求める」という計算を考える。底辺が 10cm、高さが 8cm であれば

$$\frac{10 \times 8}{2} = 40(cm^2)$$

[1] ビット (bit) は「2 進表現の 1 桁」(binary digit) から来ているが、「ちょっぴり」という意味の英語でもある。

[2] 前述の「既に知っていることを再度伝えられても情報は増えない」という観点から厳密に言えば「1 ビットのデータ」と呼ぶ方が正しいかも知れない。また、知らないことであっても「ほとんど雨が降らない地方の天気」であれば、「雨が降っていない」という知らせには新たな価値がほとんどないから情報の量としては小さいものとなる、という考え方で**情報量**を測る理論もある。

だし、底辺が 6cm、高さが 5cm であれば

$$\frac{6 \times 5}{2} = 15(cm^2)$$

だ。「電卓」で計算するのなら、実際にこれらを計算するようにキーを叩けばよい:

[1] [0] [×] [8] [÷] [2] [=]

しかし、コンピュータでの計算はこれとはちょっと違う。なぜかというと、コンピュータは非常に高速に計算ができるし、また高速に計算するためのものなので、いちいち人間が「計算ボタン」を押していたら人間の速度でしか計算が進まず意味がないからである。そのため、「どういう風に計算をするか」という「**手順**」を予め用意しておき、実際に計算するときは「**データ**」を与えてそれからその「手順」を実行させるとあっという間に計算ができる、という風になっている (もちろん、この「手順」とはプログラムのことだ)。

これを実現するためには、計算の「手順」と「データ」を分けることが必要である。たとえば面積の計算だったら、手順は

[☆] [×] [◇] [÷] [2] [=]

みたいに書いてあって、あとで「☆は 10、◇は 8」というデータを与えて一気に計算する、みたいにするわけである。もちろん、「☆は 6、◇は 5」とすれば別の三角形の計算ができる。

これを捉え直すと、「個々の三角形の面積の計算」から「具体的なデータ」を取り除いた「計算のモデル」が手順だ、ということになる。なお、モデルを作る時に「不要な側面を捨てる」ことを**抽象化**という。つまり、具体的な計算を抽象化したものが手順、という言い方をしてもよい。

ところで、コンピュータでの計算はモデル、と言うのにはもう 1 つ別の意味もある。三角形は 3 つの直線 (線分) から成るわけだが、世の中には完璧な直線など存在しないし、まして鉛筆で紙の上に引いた線は明らかに「幅」を持っていて縁はギザギザ曲がっている。また、10cm とか 8cm とか「きっかり」の長さも世の中には存在しない。でも、そういう細かいことは捨てて「理想的な三角形」に抽象化してその面積を考えて計算するわけである。

逆に言えば、コンピュータで計算する時には常に、現実世界のものをそのまま扱うわけではなく、必要な部分だけをモデルとして取り出し、それを計算している、ということになる。この意味での抽象化やモデル化には、皆さんはこれまで数学の一環として多く接して来たと思うが、これからはコンピュータでプログラムを扱う時にもこのようなモデル化を多く扱って行く。

1.4 アルゴリズムとその記述方法

前節における「三角形の面積の計算方法」のような、計算 (や情報の加工) の手順のことを**アルゴリズム** (algorithm) という。ある手順がアルゴリズムであるためには、次の条件を満たす必要がある。

- 有限の記述でできている
- 手順の各段階に曖昧さがない
- 手順を実行すると常に停止して求める答えを出す

最初の条件については、「無限に長い」記述は書くこともコンピュータに読み込ませることも不可能だから当然である。2 番目については、曖昧さがあってはそれをコンピュータで実行させるようにできないため、当然である。3 番目の条件ついてはどうだろう。実際にコンピュータのプログラムを書いてみると、手順に問題があって実行が止まらなくなることは頻繁に経験する。そのようなものはアルゴリズムとは言えない。

また、停止することを条件にしておかないと、アルゴリズムの正しさについて論じることが難しい。たとえば、「このプログラムは永遠に計算を続けるかも知れませんが、停止したときは億万

長者になる方法を出力してくれます」と言われて、それを実行していつまでも止まらない (ように思える) 時、上の記述が正しいかどうか確かめようがない。(実は無限に計算を続けるだけのプログラムでも上の記述にはあてはまってしまう。)

アルゴリズムを考えたり検討するためには、それを何らかの方法で記述する必要がある。その記述方法としてはさまざまなものがあるが、ここでは手順や枝分かれなどをステップに分けて日本語で記述する、**疑似コード**と呼ばれる方法を使う。**コード**とは「プログラムの断片」という意味であり、「疑似」というのはプログラム言語ではなく日本語を使うから、という程度の意味あいである。

たとえば、先の三角形の面積計算のアルゴリズムを疑似コードで書くと次のようになる:

- triarea: 底辺 w、高さ h の三角形の面積を返す
- s ← $\frac{w+h}{2}$。
- 面積 s を返す

以下ではこのように、何を受け取って何を行う手順 (アルゴリズム) かを明示するようにする。上の例で「返す」というのは、底辺と高さを渡されて計算を開始し、求まった結果 (面積) を渡されたところに答えとして引き渡す、というふうに考えていただきたい。

1.5 変数と代入/手続き型計算モデル

上のアルゴリズム中で次のところをもう少しよく考えてみよう。

- s ← $\frac{w+h}{2}$。

この「←」は**代入**を表す。代入とは、右辺の**式**で表された値を計算し (計算は電卓で計算するのと同じようなもの)、その結果を左辺に書かれている**変数**に「格納する」「しまう」ことを言う。つまり、「w と h を足して、2 で割って、結果を s のところに書き込む」という動作を表している。

数式であれば $s = \frac{w+h}{2}$ ならば $h = 2s - w$ のように変形できるわけだが、アルゴリズムの場合は式は「この順番で計算する」代入は「結果をここに書く」というだけの意味だから、そのような変形はできないし無意味だ、ということに注意が必要である。(しかも困ったことに、多くのプログラミング言語では代入を表すのに文字「=」を使うので混乱しやすい。)

これをモデルという立場からとらえると、式は「コンピュータの演算回路による演算」を抽象化したもの、変数は「コンピュータ内部の主記憶 (メモリ) やレジスタなどのデータ格納場所」を抽象化したもの、そして代入は「格納場所へのデータの格納」を抽象化したもの、と考えることができる。

このような、式による演算とその結果の変数への代入によって計算が進んで行くようなモデルを**手続き的**計算モデル、と呼び、そのようなモデルに基づくプログラミング言語を**手続き型言語**と呼ぶ。手続き型計算モデルは、上述のように現在のコンピュータとその動作をそのまま素直に抽象化したものになっている (そのため最も古くからある計算モデルでもある)。

コンピュータによる計算を表すモデルとしては他に、関数とその評価に土台を置く関数型モデルや、論理に土台を置く論理型モデルなどもあるが、上記のような理由手続き型モデルが今のところもっとも広く使われている。

2 アルゴリズムとプログラミング言語

2.1 プログラミング言語

「プログラム」とは、アルゴリズムを実際にコンピュータ与えられる形で表現したものであり、その具体的な「書き表し方」がプログラミング言語である。「言語」という名前はついているが、プログラミング言語は「日本語」や「英語」などの「自然言語」とは違って、あくまでコンピュー

タに読み込ませて処理できることが前提の「人工言語」であり、そのため書き方も杓子定規なのがふつうである。

プログラミング言語も、すでにご存じと思うが、さまざまな特徴を持つさまざまなものが使われている。たとえば、スクリプト言語と呼ばれる、少ない行数でコードが書けて簡単に試して見られるという特徴を持った言語族が最近広く使われるようになってきている。ここでも、この言語族に属する Ruby という言語を例題および実習に使う。Ruby 以外に、Perl、Python、JavaScript、PHP などの言語もこの種族に属している。

2.2 Ruby 言語による記述

では、三角形の面積計算アルゴリズムを Ruby プログラムに直してみる。ただし当分の間、入力と出力は `irb` コマンドの機能を使わせてもらって楽をするので、計算部分だけを Ruby の**メソッド** (method、一連の処理に名前をつけたもの) として書くことにする。というわけで、三角形の面積計算メソッドは次の通り:

```
def triarea(w, h)
  s = (w * h) / 2.0
  return s
end
```

詳細を説明しよう。

1. 「`def` メソッド名」から対応する「`end`」までがメソッドとなる。
2. メソッド名の後にかっこで囲まれた名前の並びがある場合、それらはパラメタ名となる。メソッドを呼び出すとき、これらのパラメタに対応する値を指定する。
3. メソッド内には式がいくつあってもよい。各式は行を分けて記述するか、1 行に書く場合は「`;`」で区切る。たとえば上のコードの 2・3 行目は「`s = (w * h) / 2.0; return s`」のように 1 行にしても同じことである。
4. 式は原則として先頭から順に 1 つずつ実行される。
5. 「`return` 式」を実行すると、メソッドの実行は終わり、その式の値がメソッドの値となる。

なお、上の例は擬似コードに会わせるように、面積の計算結果を変数 `s` に入れてからそれを `return` していたが、`return` の後ろに計算式を直接書くこともできるので、次のようにしても同じである:

```
def triarea(w, h)
  return (w * h) / 2.0
end
```

このように、たったこれだけのコードでも、大変細かい規則に従って書き方が決まっていることが分かる。要は、プログラミング言語というのはコンピュータに対して実際にアルゴリズムを実行する際のありとあらゆる細かい所まで指示できるように決めた形式であり、だからプログラムのどこか少しでも変更するとコンピュータの動作もそれに相応して変わるか、(もっとよくあることだが) そういう風には変えられないよ、と怒られることになっている。いくら怒られても偉いのは人間であってコンピュータではないのだから、そういうものだと思って許してやって頂きたい。

2.3 動かしてみよう!

まず、エディタ (Emacs でもテキストエディットでもそれ以外のものでも、好きなものでよい) で上と同じ内容を「`sam1.rb`」というファイルに打ち込む。Ruby プログラムを格納するファイルは最後を「`.rb`」にするというのが通例である。

次に、Terminal アプリケーションを起動してコマンドが打ち込める窓を出す。ここで **irb** コマンドを実行して Ruby 実行系を起動する。

```
% irb
irb(main):001:0>
```

この「irb なんとか>」というのは irb のプロンプトで、ここの状態で Ruby のコードを打ち込むことができる。ここでは先に作ったファイル sam1.rb を読み込ませて、メソッド triarea を定義させる。

```
irb(main):001:0> load 'sam1.rb'
=> true
irb(main):002:0>
```

true(論理値の「真」) が表示されたら成功。成功しなかった場合は、ファイルの起き場所やファイル名の間違い、ファイル内容の打ち間違いが原因と思われるのでよく調べて再度 load をやり直すこと。

なぜわざわざ 3〜4 行程度の内容を別のファイルに入れて面倒なことをしているのだろう? それは、メソッド定義の中に間違いがあったらそれを毎回 irb に向かって打ち直すのでは大変すぎるから。このため、以下でもメソッド定義はファイルに入れて必要に応じて直し、irb では load とメソッドを呼び出して実行させるところだけやる、という分担にする。

さて、load 成功したら triarea が定義できているはずなので、それを実行する。

```
irb(main):002:0> triarea(8, 5)
=> 20.0
irb(main):003:0> triarea(7, 3)
=> 10.5
irb(main):004:0>
```

確かに実行できているようだ。irb を使い終わるときは、Control-D(Control キーを押しながら D のキーを打つ) で終わらせられる。

```
irb(main):004:0> ^D     ← Control-D を打った
%
```

苦労のわりにはあんまり大したことはない感じだが、まあ初心者の第 1 歩ということで、そうがっかりしないで戴きたい。

演習 1 例題の三角形の面積計算メソッドをそのまま打ち込み、irb で実行させてみよ。数字でないものを与えたりするとどうなるかも試せ。

演習 2 三角形の面積計算で、割る数の指定を「2.0」でなくただの「2」にした場合に何か違いがあるか試せ。

演習 3 次のような計算をするメソッドを作って動かせ。なお、1 つのファイルにメソッド定義 (def ... end) はいくつ入れてもよい。

a. 2 つの実数を与え、その和を返す (ついでに、差、商、積も)。何か気づいたことがあれば述べよ。
b. 「%」という演算子は剰余を求める演算である。上と同様に剰余もやってみよ。何か気づいたことがあれば述べよ。
c. 円錐の底面の半径と高さを与え、体積を返す。
d. 円錐の底面の半径と高さを与え、表面積を返す。
e. 実数 x を与え、x を 10 で割った結果を返す。また、同様だが x の 0.1 倍を返す。これらを比較し、何か気がついたことがあれば述べよ。
f. 実数 x を与え、x の平方根を出力する。さまざまな値について計算し、何か気がついたことがあれば述べよ。

g. その他、自分が面白いと思う計算を行うメソッドを作って動かせ。

x の平方根は `Math.sqrt(`x`)` で計算できます。なお、e. や f. をやる場合は、数値を表示するときに十分な桁数がないと細かい違いが分からないだろうから、その説明をしておく。先の例のように `irb` を使って出力しているぶんには桁数などは「おまかせ」で出力されるが、自分で制御するときは出力命令を使う。とりあえず、次の 2 つを憶えておきたい。

- `puts(`値`)` — 値を (文字列でなければ) 文字列に変換し、出力する。
- `printf("`書式文字列`", `値`, `値`, …)` — 「書式文字列」を出力するが、その中に「出力指定」が埋め込まれていたら、その箇所に後ろの値を (書式に従って文字列に変換した上で) 順次埋め込んで行く。たとえば「`%.`N`g`」という出力指定は数値を有効数字 N 桁で表示、という意味であり、

  ```
  printf("%.20g %.20g\n", x, y)
  ```

  とすると、x と y の値を有効数字 20 桁で出力し、最後に改行する。

3 コンピュータ上での数値の表現

3.1 十進表現と二進表現

コンピュータが作られた最初の目的は、人間に替わって文字通り「計算」を高速に/大量に/正確に行うことだった。このため、コンピュータでもっとも最初に扱われたデータの種類は**数値**だった。

数を表現する方法としては、漢数字 (一、二、三、四、…、九、十、十一、十二、…) やローマ数字 (I、II、III、IV、…、IX、X、XI、XII、…) などもあるが、**アラビア数字** (0〜9 の数字) を用いた**位取り記法**が圧倒的に多く使われている。これは、位取り記法がなければ計算はほとんど不可能だからである。(たとえば千三百二十八から八百十三を「0〜9」で書き直さずに引き算してみれば分かる。)

我々が使う (**十進表現**ないし**十進法**の) 位取り記法では、数字として 0〜9 までの 10 種類ですべての数を書き表し、その値は桁が 1 増えるごとに十倍になる。たとえば「120」は「12」の十倍である。これは次のように説明できる:

$$1 \times 10^2 + 2 \times 10^1 + 0 \times 10^0$$
$$1 \times 10^1 + 2 \times 10^0$$

つまり、(十進表現の) 位取り記法で表された数は、左から順に $10^0 = 1$ 倍、$10^1 = 10$ 倍、$10^2 = 100$ 倍、…された値を表しているものとして扱われる。これによって、数字は 0〜9 までしかないのに、それを「並べる」ことでいくらでも大きな数が表せるわけである。

ところで、この「10」という値は特別ではなく、別の数を用いることもできる。この、位取りの基準となる数を**基数**と呼ぶ。我々が基数として「10」を使っている (十進表現を使っている) のは、単なる偶然 (指が 10 本あるから?) とされている。

これがもし「三進表現」であれば、数字として「0、1、2」の 3 種類を用い、1 桁右に行くごとに 3 倍の値を表すことになる。たとえば三進表現の「120」は次のように十進表現の「15」を表している (添字にかっこつきの数を書いて基数を表している)。

$$120_{(3)} = 1 \times 3^2 + 2 \times 3^1 + 0 \times 3^0 = 15_{(10)}$$

そして、コンピュータではおもに**二進表現** (ないし**二進法**) が使われる。これは、コンピュータの実現に使う電子回路では「電流が流れている/いない」「電圧がある/ない」など 2 つの状態を持たせる回路が作りやすいためである。[3][4]

二進表現では、数値として「0、1」の 2 種類を用い、1 桁右に行くごとに 2 倍の数を表すことになる。たとえば「$1010_{(2)}$」は次のように十進表現の 10 を表す:

$$1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 8_{(10)} + 2_{(10)} = 10_{(10)}$$

3.2 負の数の表現と二の補数

上で説明した二進表現では、N ビットの場合 0〜$2^N - 1$ までの範囲の数が表せることになる。これを (負の数が含まれないという意味で) **符号なし二進表現**と呼ぶこともある。

しかしコンピュータでの計算では、負の数も当然扱いたい。このため、1 ビットを**符号ビット**として用い、正負の数をともに扱うような表現方法が複数作られた。ここではその中から、現在ほとんどのコンピュータで採用されている**二の補数表現**について説明する。

二の補数表現とは、簡単に言えば「符号なし二進表現の上半分 (再上位ビットが 1) の範囲を、そのまま負の数の側に移したもの」と考えるとよい。たとえば、3 ビットの符号なし二進表現と二の補数の対応は次のようになっている:

値	二進	二の補数
7	111	
6	110	
5	101	
4	100	
3	011	011
2	010	010
1	001	001
0	000	000
-1		111
-2		110
-3		101
-4		100

つまり、3 ビットの符号なし二進表現では 0〜7 の範囲の値が表せるが、二の補数では -4〜3 の範囲の値が表せる。二の補数表現の特徴として、符号なし二進表現の計算と同じ回路で (単に最上位からの桁上がりを無視するだけで) 負の数を含んだ計算がそのまま行える、という点があげられる。

たとえば「$-2 + 3 = 1$」は「$110 + 011 = (1)001$」となり、確かに最上位の桁上がりを無視する点以外は符号なし二進表現と同じ計算で行えている。

また、数の符号を反転する (マイナス 1 を掛ける) 操作は、「各ビットの 0/1 を反転してから 1 を足す」操作で行える。たとえば、3 は「011」なので、その 0/1 を反転して「100」、さらに 1 を足すと「101」となり、これは確かに -3 の二の補数になっている。逆も一応示しておくと、「101」→「010」→「011」で確かに元の 3 に戻る。

符号なしの整数ついても 2 の補数表現の整数についても、整数という本来は無限個あるもののなかから、与えられたビット数で表せる有限の範囲を「切り取って来て」表現しているため、演算の結果が表せる範囲を超えてしまうと正しくない結果が得られる。具体的には「正の数と正の

[3] コンピュータの黎明期に、日本の独自技術として、3 状態を持つ回路素子を使ったコンピュータが作られたことがあり、そこでは三進表現が採用されていた。

[4] 二進表現/十進表現された数のことを**二進数**/**十進数**と呼ぶ流儀もあるが、数そのものはどのように表記しても同じなので厳密に言えばおかしい用語である。また、数学では素数 p に対する「p **進法**」という用語を全く別の意味で用いる。

数を足したのに負の数になった」等のことが起こる。このような、扱える範囲を越える演算を行ったために結果が正しくなくなることを一般に**あふれ** (overflow) と呼ぶ。

また、2 の補数ではマイナスの数は 0 以上の数より 1 個多く表せるため、「符号を反転したのにまた元の数に戻ってしまう」数が存在したりする (この場合もあふれが起きている)。コンピュータで数値を扱う時は、このようなことを常に意識しておく必要がある。

さて、以上の説明は多くのプログラミング言語 (C、C++、Java 等) にあてはまるのだが (これらでは 32 ビットの 2 の補数表現が使われる)、Ruby ではちょっと事情が違う。上のような限界はあくまでも「ビット数が決まっている」場合に起きることなのだった。これに対処するため、Ruby では整数値の演算結果がある標準のビット数以内で表せなくなった時には適宜ビット数を増やして表せる範囲を広げるようになっている。そういうわけで、Ruby では整数の限界に伴う問題にぶつかることがなくなっているが、その代わり「数が大きくなるにつれて計算に掛かる時間も多くなる」ことになるのでやはり「数学の数とは違う」という注意は必要である。

演習 4 Ruby で足し算を行うメソッドを「多数回実行させて所要時間を計る」ことを考える。このとき、足し算の結果が大きくなると、上で説明したように、あるところからはそれ以前より所要時間が増えるはずである。その「あるところ」はいくつか推定し、実際に時間計測してそのことを確かめよ。

この課題をやる場合には「時間を計る」必要があるが、そのためのメソッド定義として次のものを打ち込んで使うとよい。これは (1) まず現在の CPU 消費時間累計を取得し、次に (2) 指定した回数だけ同じ処理を反復し、(3) また現在の CPU 消費時間累計を取得して、(4)2 つの時間の差 (つまり反復した処理の分の CPU 消費時間) を返すようになっている。

```
def bench(count, &block)
  t1 = Process.times.utime
  count.times do yield end
  t2 = Process.times.utime
  return t2-t1
end
```

これを (load してから) 使う時は、次のように反復回数を指定し、また do ... end(ブロック) の間に繰り返し実行させたい処理を書く。

```
irb(main):095:0> bench 100000 do add(100000000, 1) end
=> 0.125      ← CPU 消費時間が秒単位で求まる
```

これらの仕組みの説明は長くなるので今回は省略する。

3.3 浮動小数点

ここまでは「正負の整数」を扱って来たが、数にはもちろん小数点つきの数もある。数学の世界では「整数」は「実数」の真部分集合だが、コンピュータ上の数の表現の場合は「整数」と「実数」はまったく違った性質を持っていて、プログラムの上でもきっぱり区別される。

たとえば、先の三角形の面積のプログラムで割る数を「2.0」としたのと「2」としたのでは挙動が違うのが分かっただろうか (演習 2)。これは「2.0」が実数を表す定数、「2」が整数を表す定数という違いがあるためである。数学だったらそんなものの違いは存在しない、2 は 2 に違いないわけだが、コンピュータではそうは行かない。そして「/」の演算は、分母・分子とも整数の場合は「小学校で小数点を習う前の割算」になるので、余りがあっても切り捨てられる。分母または分子の少なくとも 1 方が実数なら、「小数点を習った後の割算」になる。というわけだったのだ (だから、割る数を「2」にしても、底辺か高さのどちらかを小数点つきで入力すれば小数点つきの結果になる)。

では具体的には、有限のビット数で実数を表すのにはどうしたらいいだろうか? たとえば、10進数で 8 桁ぶんの整数を表す方法があるのなら、そのうちの下から 4 桁が小数点以下、その上が小数点以上、のように考えればそれで小数点つきの数が表せるのではないだろうか?

□□□□. □□□□

このような考え方を、小数点が決まった位置に固定されていることから**固定小数点**による実数表現と呼ぶ。しかし実際には、この方法はあまりうまく行かない。というのは、科学技術計算ではすぐに「30,000,000」だとか「0.0000001」のような数値が出て来るので、この方法ではすぐに限界になってしまうからである。

ではどうしたらいだろうか。そのヒントは、理系では上のような数値の表現ではなく、「3×10^8」とか「1×10^{-6}」のような記法が多く使われる、というところにある。つまり、1 つの数値を**指数** (桁取り) と**仮数** (有効数字) に分けて扱うことで広い範囲の数値を柔軟に扱うことができるわけである。この方法は、指数によって小数点の位置を動かすものと考えて**浮動小数点**と呼ばれる。

たとえば、同じ 10 進数 8 桁ぶんでも、6 桁の有効数字と 2 桁の指数に分けた浮動小数点表現を扱うとすれば、表せる絶対値のもっとも大きい数は「$\pm$ 9.99999 $\times 10^{99}$」、0 でない絶対値のもっとも小さい数は「0.00001 $\times 10^{-99}$」ということになり、ずっと広い範囲の数が扱えることになる。

実際にはコンピュータでは 2 進法を使うため、上と同様のことを 2 進表現で行っている。多くのプログラミング言語の実数データ型では、符号 1 ビット、仮数部 52 ビット、指数部 (符号含む)11 ビット、合計 64 ビットの浮動小数点表現が使われている (この割り当ては IEEE754 と呼ばれる規格に従ったもの)。

浮動小数点を用いた実数表現には、整数の表現とはまた違った注意点がある。まず、有効数字は当然ながら有限なので、その範囲で表せない結果の細かい部分は丸め (十進表現で言えば四捨五入) が行われて、**丸め誤差**となる (言い替えれば、コンピュータによる実数計算は基本的に近似値による計算を行っているものと考えるべきである)。

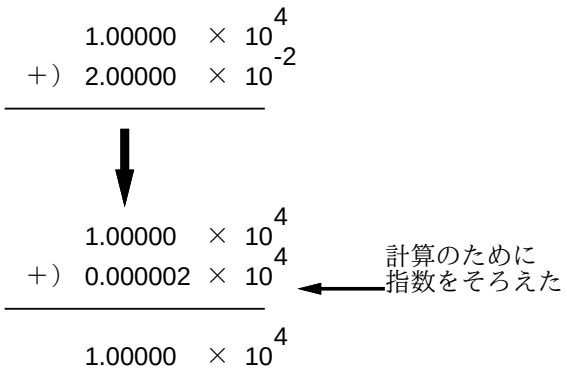

図 2: 浮動小数点演算の弱点

また、絶対値が大きく異る 2 つの数を足したり引いたりすると、絶対値が小さい方の数値の下の桁は (演算のための桁揃えの結果) 捨てられてしまうので、これも誤差の原因となる (**情報落ち**という)。極端な例として、演算した結果が元の (絶対値が大きい) 数のまま、ということも起こる。これは、たとえば図 2 のような例を思い浮かべて見れば分かる。

逆に、非常に値が近い数値どうしを引き算する場合も、上の方の桁がすべて 0 になるため、結果は元の数の下の部分だけから得られたものとなり、やはり誤差が大きくなる。これを**桁落ち**という。

なお、整数では全てのビットのパターンを数値の表現として使っていたが、浮動小数点では指数部と仮数部の組み合わせ方に制約があるので (たとえば仮数部が 0 であれば値が 0 なので指数部には意味がなく、このときは指数部も 0 にしておくのが普通)、これを利用して「+∞」「-∞」「NaN (Not a Number、非数)」などの特別な値を用意している。また、0 にも「+0」と「-0」があったりする。だから、演算の結果としてこれらのヘンな値が表示されても驚かないように。

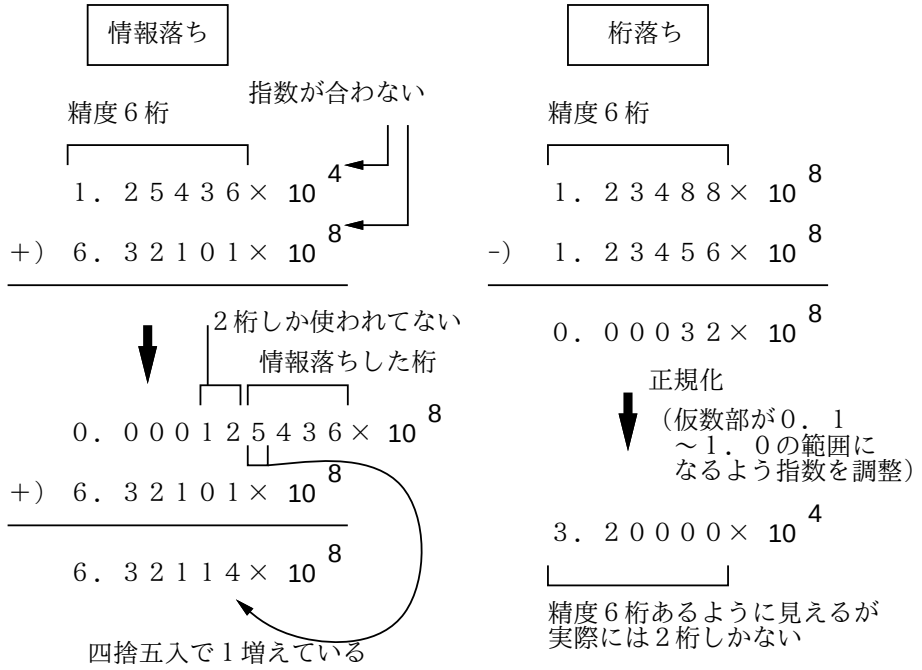

図 3: 情報落ちと桁落ち

演習 5 実数型の浮動小数点の演算で誤差が現れるような計算の例を Ruby プログラムで試してみよ。どのような場合にどのような誤差が現れるかについて考察すること。(注意! これをやる場合は先に説明した `printf("%.20g\n, 値)` などを使わないと十分な桁数の表示が行われないのでうまく検討できないと思う。)

演習 6 実数型の浮動小数点の演算で±∞、NaN、-0 などが現れるのはどんな場合かについて Ruby プログラムで試してみよ。単にどうやったらどうなっただけでなく、一般的にどうなっていると思うか考察すること。

A 本日の課題 1A

今日は「演習 3」で動かしたプログラム (どれか 1 つでよい) 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。メールアドレスは

`kuno@mail.ecc.u-tokyo.ac.jp`

です。具体的な内容は次の通り。

1. Subject: は ASCII(いわゆる半角) 文字で「Report 1A」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 「演習 3」で動かしたプログラムどれか 1 つのソース (冒頭に何のプログラムかくらいは説明をつけてください)。コピー&ペーストなどで挿入すること (エンコードされた添付ファイルはいちいち解読する手間が掛けられないので避けてください)。
4. 以下のアンケートの回答 (簡単でよい)

   Q1. プログラム、って恐そうですか? 第 2 外国語と比べてどう?

   Q2. Ruby 言語のプログラムを打ち込んで実行してみて、どのような感想を持ちましたか?

   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 1B

次回までの課題は「演習 3」の (小) 課題 (ただし 1A で提出したものは除外)、「演習 4」～「演習 6」を合わせたものから 2 つ選択してプログラムを作り考察も含めて報告すること。「演習 4」～「演習 6」のうちから最低 1 つは選ぶこと。

レポートは授業開始時までに、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 1B」とする。
2. 学籍番号、氏名、投稿日時を書く。
3. 選んだ課題プログラム 1 つのソース。
4. 説明と考察。
5. 選んだ課題プログラムもう 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. プログラムを作るという課題はどれくらい大変でしたか?
   Q2. コンピュータでの数値の計算に対する数学とは違う挙動についてどう思いましたか?
   Q3. 課題に対する感想と今後の要望をお書きください。

C　その他

注意! 自動集計する都合上、レポートのメールはすべて東大 ECC のアカウントから出してください。自宅等、別のアカウントから来たものは「保留」します (後日東大 ECC から同じものを送ってもらった時点で受理します)。よろしく。

レポートは× (提出なし)、△ (遅刻、保留、ないし内容に問題)、○ (普通)、◎ (特に買うべき点がある)、の 4 段階で評価します。課題部分の点数は全提出が○以上で満点ですので、◎は「△や×の穴埋め」に使います。さらに期末テストの穴埋めにも使うかどうかは考慮中。

「◎」の基準ですが、久野から見て「これは工夫されている/買える/よいアイデアがある」と判断したものに差し上げます。プログラムが高度ならいいとかいうものではなく、その人のレベルから見て工夫があれば買います。上記の通り、「満点を超える余興」ですので乱発する気はありません。

なお、レポートにアンケートの回答が付随していなかったり、回答として内容のないもの (例: 全項目に「よくわかりません」「?」「むずい」等の記入しかないもの) は△になると思ってください。アンケートは授業内容に関する重要なフィードバック材料ですので、簡単でいいですからちゃんと記入してください。遅刻の「△」は差し替えませんが、アンケート等の内容不完全で「△」のものは後日適切なものを再提出して頂ければ「○」に差し替えます。

その他、個人的な質問等があればいつでも、メールで久野 (上記メールアドレスです) あてお知らせください。ただしレポートと混同するような Subject:は避けてくださいね^_^;。課題の分からないところ等、全般的な質問であれば掲示板の方がよいと思います。

次回から資料は自分で打ち出してきてください。本クラスの Web ページの「資料」ページに資料の PDF 版へのリンクを起きますから、自宅でも大学ででも打ち出してください。授業時に資料を持って来てないと時間を無駄にしますから、必ず予め打ち出し、できるだけ目を通して来てください。印刷がもったいないからと画面で見るだけで済ませる人がいますが、久野個人としては「紙に打ち出して繰り返し資料を読む」ことが上達の早道だと考えています。いくら紙が節約できてもプログラミングで挫折したら大損でしょう?