情報科学　2011　久野クラス　＃1

久野　靖*

2011.10.7

はじめに — なぜ「情報科学」選択を薦めるか

こんにちは、久野です。「情報科学 (金曜 2 限)」を担当します。

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。つまり、「情報科学」と「プログラミング」を並行して学ぶことになります。実はこれはとても良いことです。というのは、プログラミングをきちんと学ぶことによって、コンピュータの (正確にはソフトウェアの) 動作がどのようなものであるか、大体分かるようになります。

世の中の多くの人は、「特定のソフトウェアの機能や操作方法」を個別に学びますが、それだと別のソフトウェアに対面した時や、悪くすると同じソフトウェアの新バージョンに対面した時に、これまでの知識が通用しないことになります。

これに対し、プログラミングから学ぶことによって、ソフトウェアに対するより一般的な「古くならない」理解力がつき、初めてのソフトウェアでも「だいたいこうだろう」という感覚で操作できるようになります。ウソみたいだと思うかも知れませんが、情報系を専門としてきた人間は実際にそうなのです。一般教養でもこの方法を活かさない手はありません。ちなみに、「情報科学」の側の知識も原理的なものであり、やはり古くはなりませんし、ソフトウェアのことがさらによく分かるようになります。

ただし、プログラミングをきちんとマスターするには、それなりに集中して学ぶ必要があります。今週の講義/演習をやって、それから 1 週間ほっておいて、翌週忘れたころに続きをやる、というのでは駄目なのです。このため、本クラスでは次の方法を取ります。

- およそ半分の 7 回 (進度の様子によっては 8 回になるかも知れません) は、毎回「授業日当日の課題 (A 課題)」と「翌週の授業日までの課題 (B 課題)」を課し、集中的に学んでもらう。
- 以後は「授業日当日の課題 (A 課題)」のみにする。
- 予定内容が終わった段階で以後は「出席自由」にする。

出席点 (内実はレポート点ですが、明白な不備がない限り満点) は、各課題ごとにつけますので、前半だけで 7 割以上の出席点となります。他科目のレポートや試験で忙しい時期に負担が軽くなるので悪くない話だと思いますがどうでしょうか。

なお、「情報科学」では全クラス共通の試験を行うので、本クラスも試験なしにはできませんが、本クラスでは配点を出席点:試験点で 50:50 としますので、それなりにやれば大丈夫のはずです。

本科目は必修ではないので、「パス」する学生さんが例年多いのですが、選択した学生さんからは例年ほぼ 100%「選択してよかった」、(未経験者の場合)「コンピュータに対する見かたが全く変わった」旨の感想を頂いています。「プログラミングなんて自分には不要だし関係ない」という誤解が多いのですが、そうではないことは冒頭で述べた通りです。そういうわけで、ぜひ本クラスを選択して、「情報科学」と「プログラミング」をマスターされることを、お勧めします。

*筑波大学大学院経営システム科学専攻

Ruby と Web

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。また、このクラスの Web サイトは

`http://lecture.ecc.u-tokyo.ac.jp/~kuno/is11/`

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。

```
@@@ 氏名: 久野　靖　←半角で!! 全角の@@@はだめ
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。

ペアプログラミング

本クラスでは今年から「ペアプログラミング」を採用します。これは次のようなものです。

- 1 つの画面の前に 2 人で座り、片方がキーボードを持ってプログラムを打ち込む。もう 1 人はそれを一緒に眺めて意見やコメントや考えを延べる。キーボードの担当者は適宜交替してもよい。

このような方法がよい理由としては、次のものがあげられます。

- プログラムを作って動かすのには多くの緻密で細かい注意が必要ですが、1 人でやるより 2 人でやる方がこれらの点が行き届き、無用なトラブルによる時間の空費が避けられます。
- プログラミングではたまたま「簡単な知識」が足りなくて、それを調べて使うまでにすごく時間が掛かることがありますが、2 人いればそのような知識を「どちらかは知っている」可能性が高まり、時間の無駄が省けます。
- プログラミング的な考え方を身に付けるには、さまざまな方面から思考したり、それを身体的な活動と結びつけることが有効です。2 人で互いに議論することで、思考が活発になり、多方面にわたるアイデアが出やすくなるため、上記のことがらに貢献します。

課題提出に際してはもちろん、2 人で作ったものですから、その 2 人については同一のプログラムを出して頂いて構いません。ただし次の条件があります。

- 提出するレポートにおいて、互いに「誰がペアであるか」を明示してあること。
- 「授業当日の課題 (A 課題)」と「翌週までの課題 (B 課題)」でペアを変更してはならず、1 人で作業してもいけない。つまり時間外もプログラミングについては 2 人で時間を合わせて作業すること。

- 推奨はしませんが、個人的な好みや都合よりペアを組まずに作業することも認めます。ただしこの場合は A 課題から 1 人で作業すること。
- ペアの相手の変更や、ペアと個人作業の間の変更は、毎週可能です (授業開始時に決めてください)。一旦決めたらそれは翌週まで維持すること (どうしても途中で変更しなければならない事情が発生したら、久野まで相談してください)。

レポートについてはあくまでも個人単位で出して頂き、個人単位で採点します。ペアで複数プログラムを作った場合、どれを提出するかは各自で選んで構いません。

前記のように、このやり方は今年が最初なので、色々様子が分からないところもありますが、とりあえずこの規則で始めてみます。なにか不都合があれば訂正の掲示を出します。

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。

- プログラムの基本概念、変数、演算、代入、数値の表現とその性質
- 基本的な制御構造とアルゴリズム、問題の性質の利用
- 制御構造の組み合わせ、データ型とデータ構造、配列
- 手続と抽象化、基本的な整列アルゴリズム、再帰手続き/ 関数、2 次元配列と画像
- より高度な整列アルゴリズム、時間計算量とその分析
- 連立方程式の数値解法、微分方程式の数値解法
- オブジェクト指向、乱数、ランダムアルゴリズム
- 動的/再帰的データ構造、表と探索、2 分探索木
- 抽象構文木と式木、言語処理系、インタプリタ、再帰下降解析
- スタック、キュー、抽象データ型、オートマトン、状態空間の探索
- 動的計画法、パターン認識、隠れマルコフモデル、ビタビアルゴリズム
- さまざまなプログラミング言語、プログラミング言語の歴史、強い型と弱い型
- グラフィカルユーザインタフェース、オブジェクト指向グラフィクス

あくまでも予定で、進度に応じて変更すると思います。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いえます。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ます。

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きましょう。**アナログ量** (analog quantity) とは、連続的に変化する値を表す量を言います。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す。

これに対し、**ディジタル量** (digital quantity) とは、とびとびに変化する値を表す量を言います。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します。

量をあらわすときの表し方にも、**アナログ表現** (analog representation) と**ディジタル表現** (digital representation) とがあります。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します。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ます。

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します。

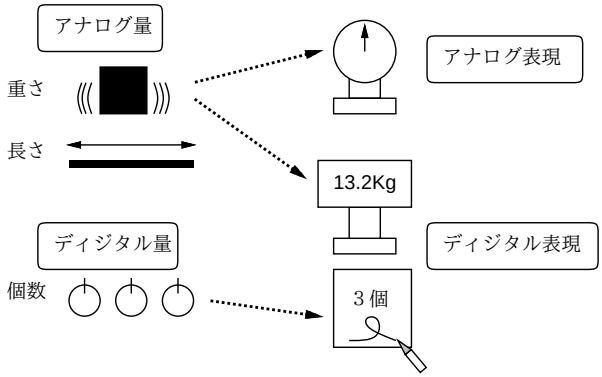

図 1: アナログとディジタル

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ります。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ませんね。

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ます。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りますが、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ります。また、遠くまでその情報を伝達するのも簡単ではありません。

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すし、数字を読み上げたりして遠くまで伝達するのも簡単です。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りません。

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**と呼ぶことにします。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ます。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」ということになります。

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ます。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ます (1t 以上の体重の人はいないでしょうから)。

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ます。これを「0/1 のどちらか」で表すこととし、「1 **ビット**の情報」と呼びます。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ます: [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ですが、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ます。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します。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ます。

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ます。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ます。

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ます。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、という点も重要です。

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います。

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、と言えます。ファッションモデルであれば、さまざまな人が服を着る、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ます (もちろんそこには服をよく見せるという意図はあるでしょうけれど)。

コンピュータで計算をするのに、なぜモデルの話をしているのでしょう? 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」だからです。たとえば、「三角形の面積を求める」という計算を考えてみましょう。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) から来ていますが、「ちょっぴり」という意味の英語でもあります。

[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しれません。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**を測る理論もあります。

ですし、底辺が 6cm、高さが 5cm であれば

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

です。「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ですね:

[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っています。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す。

具体的には、「どういうふうに計算をするか」という**手順** (procedure) を予め用意しておき、実際に計算するときは**データ** (data) 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、というふうになっているのです。そしてこの手順が**プログラム** (program) なのです。

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す。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、みたいにします。[3]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、ということになります。[4]

コンピュータでの計算はモデル、と言うのにはもう 1 つ別の意味もあります。三角形は 3 つの直線 (正確に言えば線分) から成るわけですが、世の中には完璧な直線など存在しませんし、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ます。また、10cm とか 8cm とか「きっかり」の長さも世の中には存在しません。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しているわけです。

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て、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、ということになります。この意味での抽象化やモデル化には、皆様はこれまで数学の一環として多く接してきたと思いますが、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いきます。

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) と言います。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ります。

- 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。[5]

1 番目は、「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です。2 番目は、曖昧さがあるとそれをコンピュータで実行させられないからです。3 番目はどうで

[3]もちろん、「☆は 6、◇は 5」とすれば別の三角形の計算ができますね。

[4]モデルを作る時の「不要な側面を捨てる」という作業を**抽象化** (abstraction) と言います。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わけです。

[5]実は、計算の理論の中に「答えを出すかどうか分からないが、出したときはその答えが正しい」という手順を扱う部分もありますが、ここでは扱いません。

しょうか。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も頻繁に経験しますが、そのようなものはアルゴリズムとは言えないのです。[6]

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ります。その記述方法としてはさまざまなものがありますが、ここでは手順や枝分かれ等をステップに分けて日本語で記述する、**擬似コード** (pseudocode) と呼ばれる方法を使います。**コード** (code) とは「プログラムの断片」という意味で、「擬似」というのはプログラミング言語ではなく日本語を使うから、と考えておいてください。

三角形の面積計算のアルゴリズムを擬似コードで書いてみます: [7]

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

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ましょう:

- s ← $\frac{w \times h}{2}$。

この「←」は**代入** (assignment) を表します。代入とは、右辺の**式** (expression) [8]で表された値を計算し、その結果を左辺に書かれている**変数** (variable — コンピュータ内部の記憶場所を表すもの) に「格納する」「しまう」ことを言います。つまり、「w と h を掛けて、2 で割って、結果を s のところに書き込む」という**動作** (action) を表していて、数式のような定性的な記述とは別物なのです。

数式であれば $s = \frac{w \times h}{2}$ ならば $h = \frac{2s}{w}$ のように変形できるわけですが、アルゴリズムの場合は式は「この順番で計算する」というだけの意味、代入は「結果をここに書き込む」というだけの意味ですから、そのような変形はできないので注意してください。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、普通の数式であるかのような混乱を招きやすいのです。

これをモデルという立場からとらえると、式は「コンピュータ内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** (main storage) ないし**メモリ** (memory) 上のデータ格納場所」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動作」を抽象化したもの、と考えることができます。

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いくようなモデルを**手続き型計算モデル** (procedural computational model) 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**命令型言語** (imperative language) ないし**手続き型言語** (procedural language) と呼びます。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ます。このため手続き型計算モデルは、最も古くからある計算モデルでもあるのです。

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のですが、上記のような理由から、手続き型モデルが今のところもっとも広く使われています。

---

[6]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くなります。たとえば、「このプログラムは永遠に計算を続けるかもしれませんが、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) とき、上の記述が正しいかどうか確かめようがありません。

[7]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します。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ください。

[8]プログラミングで言う式とは、計算のしかたを**数式** (mathematical expression) に似た形で記述したものを言います。先に説明した、電卓で計算する手順を記したようなものと思ってください。

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」ないし「規則」のことを**プログラミング言語** (programming language) と呼びます。これはちょうど、人間が会話をする時の「話し方」として「日本語」「英語」などさまざまな言語があるのと同様です。ただし、**自然言語** (natural language — 日本語や英語など、人間どうしが会話したり文章を書くのに使う言語) とは違って、プログラミング言語はあくまでもコンピュータに読み込ませて処理することがが前提の人工的な言語であり、そのため書き方も杓子定規です。

ひとくちにプログラミング言語といっても、実際にはさまざまな特徴を持つ多くのものが使われています。ここでは、プログラムが簡潔に書けて簡単に試して見られるという特徴を持つ、**Ruby** という言語を用いてゆきます。

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ましょう。本クラスでは入力と出力は基本的に **irb コマンド** (irb command) [9]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) [10]として書くことにします。先にアルゴリズムを示した、三角形の面積計算を行うメソッドは次のようになります:

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

詳細を説明しましょう。

1. 「`def` メソッド名」～「`end`」の範囲が 1 つのメソッド定義になる。
2. メソッド名の後に丸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** (parameter) [11]の名前となる。メソッドを呼び出す時、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**文** (statement) [12]がいくつあってもよい。それぞれの文は行を分けて記述するか、1 行に書く場合は「`;`」で区切る。たとえばこのメソッド本体は「`s = (w * h) / 2.0; return s`」のように 1 行にしてもよい。
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 文** (return statement)「`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。

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` していましたが、`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す:

[9]Ruby の実行系に備わっているコマンドの 1 つで、さまざまな値をキーボードから入力し、それを用いてプログラムを動かす機能を提供してくれます。

[10]メソッドは他の言語で言う**手続き**ないし**サブルーチン** (subroutine) に相当し、一連の処理に名前をつけたもののことです。なお、手順も英語では procedure ですが、日本語では手順と言う場合は抽象的な (プログラムとして書き下す前の) ものを指し、手続きと言う場合はプログラムに含まれる名前のついたひとまとまりのコードを指すというふうに使い分けられます。

[11]メソッドを使用するごとに、毎回異なる値を引き渡して、それに基づいて処理を行わせるための仕組みです。

[12]プログラムの中の個々の命令のことを、プログラム言語の用語では文と呼びます。

```
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ります。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なのです。

そのため、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、(もっとよくある場合として) そういうふうには変えられないよ、と怒られることになります。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で、そういうものだと思って許してやってください。

2.3 動かしてみよう!

では、このコードを動かしてみましょう。まず、エディタで上と同じ内容を `sample1.rb` というファイルに打ち込んで保存してください。この、人間が打ち込んだプログラムを (プログラムを動かす「源」という意味で)「**ソース**」「**ソースコード**」などと呼びます。Ruby のソースファイルは最後を「`.rb`」にするというのが通例です。

例年、ここで「エディタって何?」となる人がいますので、簡単な方法を説明します。既にエディタを使っている人は無視してください。Mac OS では「TextEdit」「テキストエディット」と呼ばれるエディタがいちばん説明なしに操作できるのでこれを説明します。まず Finder の窓を出し、「アプリケーション」フォルダを選んで、その中から上記エディタをドラグしてドックに入れてください。以後はドック内のアイコンを選択することでエディタが起動できます。そうしたらプログラムを打ち込んでください。なお、プログラムの記述に際して日本語文字は当面使わないこととしてください。

重要! テキストエディットを使う場合は「フォーマット」メニューで「標準テキストにする」を選んでから打ち込み始めること。標準テキストでないと `irb` は正しく読めません。

次にエディタでファイルを打ち込んだあと、それを「どこに」保存するかも大切です。以下でコマンドを実行しようとするときには「ホーム」のファイルが直接見えるので、一番簡単なのは「ホーム」に保存することですが、もっと別の場所に整理する流儀の人はそれなりにどうぞ。あと、ファイル形式が「プレーンテキスト」である必要があります。TextEdit で保存時にこれが選べない場合は、「フォーマット」を適宜設定してください (分からなければ質問してください)。

次に、「ターミナル」のプログラムを起動して、コマンドが打ち込める窓を出します。これも、ドックに入っていない人はファインダを使ってドックに入れておくことを勧めます。そしてターミナルの窓の中で **irb** コマンドを実行して Ruby 実行系を起動してください (「`%`」はプロンプト文字列のつもりなので打ち込まないでください):

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

この「`irb` なんとか`>`」というのは `irb` の**プロンプト** (prompt — 入力をどうぞ、という意味の表示) で、ここの状態で Ruby のコードを打ち込めます。

プロンプトの読み方を説明すると、`main` というのは現在打ち込んでいる状態がメインプログラム (最初に実行される部分) に相当することを意味しています。次の数字は何行目の入力かを表しています。最後の数字はプログラムの**入れ子** (nesting — 「はじめ」と「おわり」で囲む構造の部分) の中に入るごとに 1 ずつ増え、出ると 1 ずつ減ります。とりあえずあまり気にしなくてよいでしょう。以後の実行例では見た目がごちゃごちゃしないように「`irb>`」だけを示すことにします。

次に load(ファイルからプログラムを読み込んでくる、という意味です) で sample1.rb を読み込ませます。ファイル名は**文字列** (string) として渡すので'' または""で囲んでください: [13]

```
irb> load 'sample1.rb'
=> true
irb>
```

true が表示されたら読み込みは成功で、ファイルに書かれているメソッド triarea が使える状態になります。成功しなかった場合は、ファイルの置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 をやり直してください。

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でしょうか? それは、メソッド定義の中に間違いがあった時、定義を毎回 irb に向かって打ち直すのでは大変すぎるからです。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を行う、という分担にします。

load が成功したら triarea が使えるはずなので、それを実行します:

```
irb> triarea 8, 5
=> 20.0
irb> triarea 7, 3
=> 10.5
irb>
```

確かに実行できているようです。irb は quit で終わらせられます:

```
irb> quit
%
```

苦労のわりにはあんまり大したことはない感じでしたが、まあ初心者の第 1 歩ということで、着実に進んでいきましょう。

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。[14]

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** (reminder) 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. 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。
e. 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。[15]

[13] 本来ならメソッドに渡すパラメタは丸かっこで囲むのですが、Ruby では曖昧さが生じない範囲でパラメタを囲む丸かっこを省略できます。本資料ではプログラム例の丸かっこは省略しませんが、irb コマンドに打ち込む時は見た目がすっきりするので丸かっこを適宜省略します。

[14] 1 つのファイルにメソッド定義 (def ... end) はいくつ入れても構わないので、ファイルが長くなりすぎない範囲でまとめて入れておいた方が扱いやすいと思います。

[15] x の**平方根** (square root) は **Math.sqrt**(x) で計算できます。

f. 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。

e や f をやる場合は、数値を表示する時に十分な桁数がないと細かい違いが分からないので、そのための出力命令の説明をしておきます。先の例のように irb を使って自動的に出力させる場合は、桁数などは「おまかせ」になりますが、これを自分で制御する時は出力命令を使う必要があるわけです。

- 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) — 「書式文字列」 を出力しますが、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** (format) に従って文字列に変換した上で順次埋め込みます。とくに「%.Ng」という出力指定は数値を有効数字 N 桁で表示する指定です。たとえば次のようにすると、x の値を有効数字 20 桁で出力し、最後に改行します:

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

注意! 表示の桁数はものすごく (50 桁くらいまで?) 多くできますが、コンピュータ内部での実数 (小数点つき数) の計算は「ある決まった精度まで」でしか行われません。そのことを確認するのも課題のうちだと思ってください。ちょっと irb で直接確認してみましょう。

```
irb> printf "%.20g\n", 1.0/3.0
0.33333333333333331483
=> nil
irb> printf "%.30g\n", 1.0/3.0
0.333333333333333314829616256247
=> nil
```

表示の桁数だけ増やしても計算の精度は変わらないことが分かると思います (じゃあ、どれくらいの精度なのかな?)。なお、整数についてはこの限りではありません (こちらも、どう限りでないかを確認することも課題のうちだと思ってください)。

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当時の主要な目的は、人間に代わって文字通り「計算」を高速に/大量に/正確に行うことでした。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** (numerical value) でした。

数を表現する方法としては、**アラビア数字** (Arabic numerals — 0〜9 の数字) を用いた**位取り記法** (positional notation) が圧倒的に多く使われています。(一方、位取り記法を使わない表現方法として、**漢数字** (Chinese numerals)(一、二、三、四、…、九、十、十一、十二、…) や**ローマ数字** (Roman numerals)(I、II、III、IV、…、IX、X、XI、XII、…) などがあります。)

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す。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ると、位取り記法が計算のために不可欠だということが納得できると思います。

私達が使う**十進表現** (decimal representation) ないし**十進法** (decimal system) 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ります。たとえば「120」は「12」の十倍です。これは次のように説明できます:

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

つまり、(十進表現の) 位取り記法で表された数は、右から順に $10^0 = 1$ 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ます。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す。

ところで、「10」という値は特別ではなく、別の数を用いることもできます。この、位取りの基準となる数を**基数** (radix) と呼びます。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 (両手の指を合わせると 10 本あるから) なのです。

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ります。たとえば三進表現の「120」は次のように十進表現の「15」を表しています: [16]

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

そして、コンピュータでは主に**二進表現** (binary representation) ないし**二進法** (binary system) が使われます。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す。[17]

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ります。たとえば「$1010_{(2)}$」は次のように十進表現の 10 を表します:

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

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ます。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** (unsigned binary representation) と呼ぶこともあります。

しかしコンピュータでの計算では、負の数も当然必要です。このため、1 ビットを**符号ビット** (sign bit) 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ました。ここではその中から、現在の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** (two's complement representation) について説明します。

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」と考えるとよいでしょう。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図 2 のようになっています。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のに対し、二の補数では -4〜3 の範囲の値が表せます。

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ます。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ます。

また、**符号反転** (negation — マイナス 1 を掛けること) の操作は、「各ビットの 0/1 を反転してから 1 を足す」操作で行えます。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ます。逆も一応示しておくと、「101」→「010」→「011」で確かに元の 3 に戻ります。

符号なしの整数についても、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「切り取ってきて」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ことになります。具体的には

[16] 本資料では添字にかっこ付きの数を書いた場合は基数を表すものとします。

[17] 二進表現/十進表現された数のことを**二進数** (binary numbers)/**十進数** (decimal numbers) と呼ぶ流儀もありますが、数そのものはどのように表記しても同じ数なはずなので、これは厳密に言えばおかしい言葉づかいです。また、数学では素数 p に対する「p **進数** (p-adic number)」という用語を全く別の意味で用いています。

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

図 2: 3 ビットの二の補数表現

「正の数と正の数を足したのに負の数になった」などのことが起こります。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) と呼びます。

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」数が存在することになります (この場合も符号反転時にあふれが起きます)。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ります。

さて、以上の説明は多くのプログラミング言語 (C、C++、Java など) にあてはまるのですが (これらの言語では主に 32 ビットの 2 の補数表現が使われています)、Ruby ではちょっと事情が違います。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でした。これを克服するため、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自動的に広げるようになっています。このため、Ruby では整数の限界に伴う問題にぶつかることがなくなりますが、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す。

3.3 浮動小数点

ここまでは「正負の整数」を扱ってきましたが、数にはもちろん小数点付きの数もあります。数学の世界では**整数** (integral number) は**実数** (real number) の特別な場合として含まれるわけですが、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ます。

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に気づいた人もいるかと思います。ちょっと irb で確認してみましょう。

```
irb> printf "%.30g\n", 10/3.0
3.33333333333333348136306995002
=> nil
irb> printf "%.30g\n", 10/3      ←両方とも整数だと
3                                ←切捨ての割り算になる
=> nil
irb> printf "%.30g\n", 10.0/3    ←片方が実数なら
3.33333333333333348136306995002 ←実数の割り算になる
```

```
=> nil
```

これは「2.0」「3.0」が実数を表す定数、「2」「3」が整数を表す定数という違いがあるためです。数学であればそのような違いはなく、2 は 2 というだけなのですが、コンピュータでは整数と実数で扱いが違ってきます。具体的には、割り算「/」は分母・分子とも整数の場合は「小学校で小数を習う前の割算」になるので、余りがあっても切り捨てて答えが出されます。一方、分母または分子の少なくとも 1 方が実数なら、「小数を習った後の割算」になります。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付きで入力すれば小数点付きの結果になります。

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よいでしょうか? 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付きの数が表せる、という考えもあります:

□□□□. □□□□

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** (fixed point) による実数表現と呼びます。しかし実際には、この方法はあまりうまくいきません。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くるので、この方法ではすぐに限界になってしまうからです。

ではどうしたらよいでしょうか。そのヒントは、理科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、というところにあります。つまり、1 つの数値を**指数** (exponent — 桁取り) と**仮数** (mantissa — 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ます。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** (floating point) と呼ばれます。

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「0.00001×10^{-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わけです。

注意! コンピュータでは「小さい字」が使えないので、伝統的に指数部分を「e ±指数」で表します (e は exponent の e)。たとえば「3.0×10^{22}」であれば「3.0e+22」です。このような表示は「エラー」とかではないのでそのつもりで。

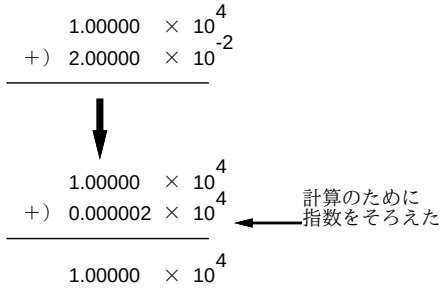

図 3: 浮動小数点演算の弱点

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 進表現で行っています。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ます。(このビットの割り当ては、**IEEE754** と呼ばれる標準に従ったものです。)

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ります。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** (rouding — 十進表現で

言えば四捨五入) が行われて、**丸め誤差** (roundoff error) となります。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なのです。

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、絶対値が小さいほうの数値の下の桁は (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ります。これを**情報落ち** (loss of information) と言います。極端な例として、演算した結果が元の (絶対値が大きいほうの) 数のまま、ということも起こります。これは、たとえば図 3 のような例を思い浮かべてみれば分かると思います。

逆に、非常に値が近い数値どうしを引き算する場合も、上のほうの桁がすべて 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ります。これを**桁落ち** (cancellation) と言います。

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ましたが、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、この時は指数部も 0 にしておくのが普通)、これを利用して正負の**無限大** (infinity — $\pm\infty$) や**非数** (NaN — Not a Number) などの特別な値を用意しています。また、0 にも「+0」と「−0」があったりします。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してください。

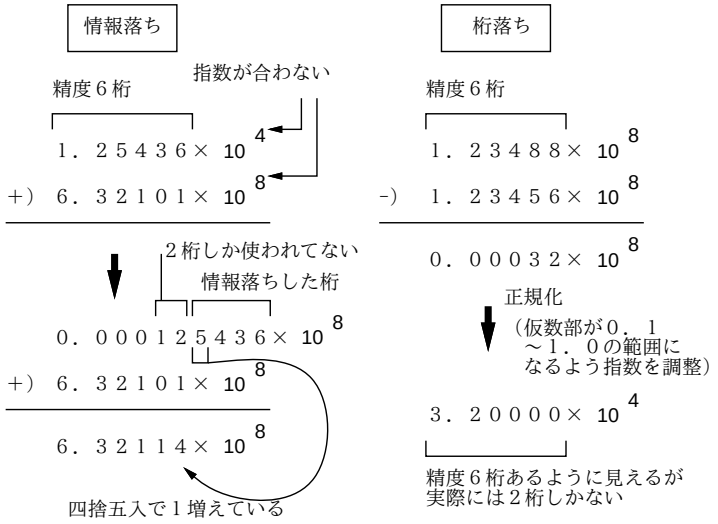

図 4: 情報落ちと桁落ち

こういうのを実際に試すのには、たとえば次のような Ruby プログラムを作ってみれば分かります。

```
def test
  printf("%.40g\n", 100000000000000000000000000000000.0 + 1.0)
end
```

これを動かしたようすは次の通りです。

```
irb> test
100000000000000000198846248386560
=> nil
```

明らかにヘンだ、と思いますか? コンピュータによる実数計算って実はこういうものなのです。

演習 4 整数の計算と実数の計算で「切捨て除算による違い以外に」結果が違うような計算の例を Ruby で試してみよ。どのような場合に違いが現れるかについて考察すること (この演習を

やる場合は、実数を十分な桁数表示しようと思った場合に先に説明した `printf` を使う必要があると思います)。

ヒント:「123451234512345 + 1」は「123451234512346」ですね。では「123451234512345.0 + 1.0」はどうでしょうか。また「12345」をもう 1 回ふやすとどうでしょうか。なぜでしょうか。

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 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 (この演習をやる場合は、先に説明した `printf` 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ません)。

ヒント: 先の例で 0 がいくつまでなら正しい答が出るか。または、1.0 にある数 ε を足しても結果が 1.0 のままであるような ε がありますが。具体的に ε はいくつか。など。

ヒント: たとえば、「ある数に 0.1 を掛ける」場合と「ある数を 10.0 で割る (10 ではなく 10.0 にすること!)」場合では結果が違うような数があります。ところが、「0.125 を掛ける」のと「8.0 で割る (8 ではなく 8.0 にすること!)」の場合は違いはありません。なぜでしょうか。

演習 6 実数型の浮動小数点の演算で $\pm\infty$、NaN、-0 などが現れるのはどのような場合かについて Ruby 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。

ヒント: たとえば 1.0/0.0 は無限大ですね。

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 (久野が常識的に認識できる程度のゆれ—大文字小文字の違いなど— は差し支えない)。
2. 学籍番号、氏名、ペアの氏名 (または「個人作業」)、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 (10/22) 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「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。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、ペアの氏名 (または「個人作業」)、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— これが「満点」)、◎ (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」ととくに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)、ペアの無断変更などは△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?