プログラミング演習

はしもとじょーじ *

2020 年 9 月 17 日

目次

* 岡山大学理学部地球科学科，e-mail: george@gfd-dennou.org

第 I 部
はじめに

1　授業の目的

1.1　コードを書く理由

何のためにコード[1]を書くのか？　手作業ではできないことをやるため，という答えはもちろんありなのだが，手作業の方が簡単にできることであっても，できるだけコードを書いて計算機にやらせる方がいいと，個人的には思っている．例えば，n 個の 2 桁の数字の和を計算するという作業があったとする．$n = 10$ だったらコードなど書かずに電卓を叩いて計算してしまう方が圧倒的に簡単であるが，それでもコードを書いた方がよいと思うのである[2]．

理由は「手作業は信用できないから」である．再現性に問題があると言うのがよいかもしれない．同じことをやっているつもりでも，手作業はやるたびに答えが違うということがよくある[3]．やり直しなどしたくない面倒な作業であるほど，答えが一致しない確率は高くなる．再現できないというのは，科学において致命的である[4]．コードを書いて処理しているなら，やるたびに答えが変わるということはない[5]．手作業でできることであっても，安心感を得るためにコードを書いて処理するのがよいと思うのである．

研究をしていると，1 回だけだったはずの作業を何度も繰り返しやり直さなければならなくなる，というのはよくあることだったりする[6]．手作業でやっていると，またやり直すのかと思うだけでうんざりしてしまうが，コードを書いて処理していれば，やり直しのハードルはずっと低くなる．すぐやり直しができるということは，研究を進める上で大きな利点になる．作業の内容がコードを書く労力に見合うかどうかは常に考えるべきである

---

[1] プログラミング言語を用いて記述されたものをコードと呼ぶ．機械語 (コンピュータが直接解釈実行可能) で記述されたものをプログラムと呼ぶ．コードとプログラムを区別しないことも多く，コードをプログラムと呼ぶことは多い．

[2] $n = 2$ のときにコードを書くか，と言われたら書かないような気がする．閾値は曖昧である．知り合いの天文屋さんは「100 までは手作業，100 を超えたらコードを書く」と言っていた．彼が業界標準なのかどうかは知らない．

[3] $n = 10$ だったら間違えるはずがないという人もいるかもしれない．そういう人は n に 10 よりも大きい数を入れてください．

[4] そう考えない人も世の中にはいるかもしれない．

[5] 「自分 (他人) が書いたコードを信用してよいのか？」という問題はもちろんあるのだが，ここではそのことに深入りしない．

[6] 個人的にはよくあることだが，万人に当てはまるかどうかは知らない．

が，後で振り返ってみたときにコードを書いてよかったと思うことは少なくないと思う[7]．

もうひとつの理由は「プログラミング[8]は楽しいから」である[9]．授業でこういうことを言う人はほとんどいないような気がするが，個人的には「楽しい」が最大の理由であると断言する．プログラミングして，実行して，思い通りに動作して，結果がでる．プログラミングによって得られる達成感は，プログラミングした人に与えられる最高のご褒美であると思う[10]．

1.2 ふたこぶラクダ

「ふたこぶラクダ」という名前で知られている有名な論文がある [1]．「ふたこぶ」というのは，プログラミング学習者の成績をグラフ化すると，高成績と低成績の２つの山が現れることを表わしている．２つの山は，学習者集団の年齢，性別，教育レベルに依らず，いつも現れるということが経験的に知られている．そして，学習者がどちらの山に入るかは，あらかじめ「一貫したメンタルモデル」を持っているかどうかで決まる，ということである．簡単にまとめると，世の中にプログラミングを理解できる人間とそうでない人間がいて，理解できない人にはいくら教えても無駄になる可能性が高い，ということらしい．

「ふたこぶラクダ」の主張が正しいかどうかの結着はついていない[11]．学校のテストの結果がふたこぶになることはよくあることで，ふたこぶは「勉強した人としなかった人」で説明されることが多い．プログラミング学習者のふたこぶ分布も，「プログラミングを理解できる人とそうでない人」ではなく，「勉強した人としなかった人」で説明されるのかもしれない．

「ふたこぶラクダ」の主張が正しいかどうかはおいておいて，プログラミングとどう向き合っていくかについて考えるためにも，まずはプログラミングをやってみるべきであると思う．やってみて，プログラミングが苦痛でしかないとしたら，プログラミングをしないで済ませることを真剣に考えるのがよいかもしれない[12]．プログラミングを楽しいと思

---

[7] 個人的な感想です．

[8] プログラムを作成する作業をプログラミングと言う．プログラミングにはコードを書く以外の作業も含まれるはずだが，プログラミング＝コードを書く，という意味で使われることが多い．

[9] ”普通のプログラマは生活のためにコードを書く．素晴らしいハッカーにとっては，コード書きは楽しみのためにするもので，それに金を払ってくれる人がいれば大いに喜ぶのだ．” (Paul Graham (著)，川合史朗 (訳) 『ハッカーと画家 コンピュータ時代の創造者たち』オーム社)

[10] 個人的な意見です．

[11] 「ふたこぶラクダ」の著者はその主張を取り下げる論文 [2] を書いているが，その論文には「実験に誤りはない」「実験には再現性がある」とも書いてある．実験結果は正しいが，その解釈はひとつに決められない，ということであるらしい．

[12] 「プログラミングをしないで済ませる」は，この授業の外での話である．この授業の到達目標は「コードを書けるようになる」であるから，単位取得のためにはプログラミングをしないで済ませることはできな

えるなら，そっちの方向を目指すのもよいかもしれない．優れたプログラマーは常に不足しているので，良いコードが書けるなら引く手数多である．

1.3 授業の方針

この授業は初めてコードを書く初心者向けのものであり，上級者向けではない．この授業では自分の手を動かして，(1) コードを書いて計算してみる，(2) 計算した結果を描いてみる，この 2 つをやってもらう．初心者向けであるから，難しいことはやらない．細かいこともできるだけ避ける[13]．この授業を履修しただけで，何かすごいことができるようになる，ということは (たぶん) ない．この授業の位置づけは「最初の一歩」である．そこから先に行く人は，自助努力によってなんとかしてもらいたい[14] [15]．

授業の基本精神は「自力更生」[16]である．授業ではいくつかのサンプル・コードを提示してその解説をするが，解説を聞くだけでは理解できないだろうと思う．解説はサンプル・コードを解読するためのヒントだと思って，各自それぞれサンプル・コードの解読を試みて欲しい[17]．やることは簡単で，サンプル・コードを読み，処理の流れを想像し，結果を予想する[18]．サンプル・コードを実行し，結果を確認する．そして，自分の理解が正しいことを確認するため，サンプル・コードを修正して実行し，結果を確認する．この作業を何度かくり返す．これだけである．これだけであるが，これをやれば練習問題のコードを自分で書くことができるようになるはずである．自分でコードを書いて，実行してみて欲しい．

この授業では Fortran と gnuplot を使用する．Fortran と gnuplot が選ばれたのは，

い．プログラミングをしないという選択をするなら，単位取得はあきらめるしかない．

[13] 必要になったときに勉強してください．

[14] “Peter Norvig は，何かでエキスパートになるには約 1 万時間の訓練が必要と述べている．” (Kevlin Henney (編)，和田卓人 (監修)，夏目大 (訳)『プログラマが知るべき 97 のこと』オライリー・ジャパン)

[15] 大学で 1 単位に相当する学修時間は 45 時間である．1 万時間を単位数に換算すると約 222 単位．これは大学を 2 回卒業できるくらいの時間に相当する．

[16] 「自力更生」とは，毛沢東が 1945 年に打ち出した政治方針．他の力に頼るのではなく，自分の力でなんとかすること．

[17] “プログラミングが上達する秘訣はコードを読むこと，そしてコードを書く事，これに尽きます．”(増井敏克『プログラマ脳を鍛える数学パズル』翔泳社)

[18] “「ええとですね，村木先生によりますと …… 自分がコンピュータさんになったつもりになって，アルゴリズムを実行するのがよいのだそうです．

- 私はコンピュータだ，と唱える．
- アルゴリズムと入力が与えられたぞ，と考える．
- そして，手続きを一歩一歩，愚直に実行していく．

…… めんどうに思えても，そのようにするのがアルゴリズムを理解する最もいい方法なのだそうです」” (結城浩『数学ガール/乱択アルゴリズム』SB クリエイティブ)

教える側の能力と環境の制約による[19]．サピア＝ウォーフ仮説[20]を認めるなら，どの言語を選ぶのかはとても重要である．世の中には様々なプログラミング言語があり，Fortran よりも優れたプログラミング言語が存在することを認めることに，やぶさかではない．この授業の後，それぞれがステップアップしていくときに，好きな言語を選んでもらったらよいと思う．プログラマーは複数のプログラミング言語を操る多言語話者であることがほとんどなので，プログラミングをしていれば複数のプログラミング言語に触れる機会が必ずあるはずである[21]．

なお，授業は Unix コマンドラインの使用を前提とする．Unix についての手取り足取りの解説はしないので，必要に応じて自習して欲しい．教室で教員から教えられるより，自分で手を動かしながら覚える方が効率がよいはずである．Unix の基礎については付録 A に少し書いたので，最低限の情報はそこで得られるはずである．

あと，授業ではシェルスクリプトも取り扱う．シェルスクリプトは，Unix シェルで用いられるスクリプト言語である．Unix シェルが解釈することのできるコマンドを並べて書いておくと，いちいちコマンドを打たなくても済むという，便利なものである．シェルスクリプトには，変数，分岐，繰り返し，といった制御構造を持たせることもできる．シェルスクリプトを使えるようになると，いろいろな作業を自動化できるようになり，作業効率が上がるはずである．

日本の伝統的な教育は自律階層的で，ひとつ上のレベルにいる人がひとつ下のレベルにいる人に教える[22]，という形になっている[23]．これは，直接的には「下位にある人のレベルを上げる仕組み」であるが，その実「上位にある人のレベルを上げる仕掛け」になっている．人に教えることは，自分の知識を確かなものとする効率のよい方法のひとつである．よく言われるのは，初学者は初級者の教材，初級者は中級者の教材，中級者は上級者

---

[19] うちの業界では，Fortran が過去に広く使われていて，現在も使われている，Fortran が劣った言語であったとしても，周囲に使っている人がいることは Fortran を使う理由になるかもしれない．それ以外に Fortran を使う理由はないかもしれない．

[20] 言語が考え方や認識を決めるという仮説．言語相対論の仮説とも言う．

[21] 個人的には面倒なことはやりたくないので，必要に迫られるまで新しい言語に手を出したりはしない．必要に迫られたときは，「言語を選ぶ」ではなく「言語を選ばされる」になることが多い．こういうタイプの人には，好きな言語を選ぶ機会など永遠に訪れないような気がする．あるプロジェクトで一緒に仕事をした人は，「プログラマーたるもの，必要があるとか役に立つとかに関係なく，毎年 1 つずつ新しいプログラミング言語を身につけるものだ」と言っていた．プログラマーを名乗ってよいのは，そういう人なのだろうと思う．

[22] 自分が学生だった頃，ある授業で教官に質問したら「それは TA(ティーチング・アシスタント) に訊いてください」と言われて相手にしてもらえなかった．純真な学生さんは教員に門前払いされると傷つくかもしれないが，日本伝統的教育の仕組みを理解すれば，傷つくようなことではないと分かるだろう．

[23] 日本伝統的教育を実践している場は少なくなっているらしいので，ここは「なっていた」と過去形で語るべきかもしれない．

の教材，以下略，である．教える/教わることで，自分も相手もレベルアップすることができる．隣にいる人が困っていて自分に教えられることがあるなら，教えてあげるとよい[24]．

GNU Linux

授業では GNU/Linux を使う．

Linux は，1991 年にヘルシンキ大学の学生 Linux Torvalds (リーヌス・トーバルズ) 氏によって開発された，UNIX と同様の動作をする UNIX 互換 OS である．オープンソース・ソフトウェアであり，いくつかの条件のもとで，使用，複製，改変，再配布が認められている．Linux は，狭義にはカーネルのみの呼称である．カーネルは様々なアプリケーションとセットで使用される．カーネルと各種アプリケーションを束ねたものは，ディストリビューションと呼ばれる．Linux のディストリビューションには，Debian GNU/Linux, Ubuntu, Knoppix, Fedra Core, Red Hat, などがある．

GNU/Linux に含まれる多くのコードは Richard Stallman (リチャード・ストールマン) 氏の GNU プロジェクトから来ている．GNU プロジェクトは，フリー (自由) でないソフトウェアを全く使わないでも済む[a]ように，十分な自由ソフトウェア (フリーソフトウェア) を開発することを目標とするプロジェクト．GNU は “GNU’s not Unix!”，再帰的頭字語[b]である．「GNU は Unix ではない」と名乗る通り，Unix 系の設計であるが，UNIX に由来するソースコードは使っていない．GNU は「グヌー」と発音する[c]．

[a] ストールマン氏が大学院生だったとき，MIT AI ラボの共用のプリンタのプリンタドライバを貰おうとしたら「それはライセンス違反だ」と言われたことがきっかけとなって，ストールマン氏は「自由ソフトウェア」運動を始めた (という話である)．わたしたちが使うソフトウェアが自由でなければならない理由は，リチャード・ストールマン「GNU プロジェクト」 `https://www.gnu.org/gnu/thegnuproject.ja.html`

[b] 再帰的頭字語とは，名称の中にそれ自身が含まれている頭字語．

[c] “The name “GNU” is a recursive acronym for “GNU’s Not Unix.” “GNU” is pronounced g’noo, as one syllable, like saying “grew” but replacing the r with n.” (Free Software Foundation “What is GNU?” `http://www.gnu.org/home.en.html`)

[24] 教えられたくない人に無理やり教えるのはやめた方がよいと思う．

自由ソフトウェア (free software)

GNU プロジェクトは「自由ソフトウェア」の定義 (https://www.gnu.org/philosophy/free-sw.ja.html) をおこなっている．以下，GNU プロジェクトによる「自由ソフトウェア」の定義の一部を引用する．

> 「自由ソフトウェア」は利用者の自由とコミュニティを尊重するソフトウェアを意味します．おおよそで言うと，そのソフトウェアを，実行，コピー，配布，研究，変更，改良する自由を利用者が有することを意味します．ですから，「自由ソフトウェア」は自由の問題であり，値段の問題ではありません．この考え方を理解するには，「ビール飲み放題 (free beer)」ではなく「言論の自由 (free speech)」を考えてください．わたしたちは無償の意味ではないことを示すのに時々，「自由」を表すフランス語あるいはスペイン語の言葉を借りて「リブレソフトウェア」と呼びます．
>
> 四つの基本的な自由
>
> あるプログラムが自由ソフトウェアであるとは，そのプログラムの利用者が，以下の四つの必須の自由を有するときです：
>
> - どんな目的に対しても，プログラムを望むままに実行する自由 (第零の自由)．
> - プログラムがどのように動作しているか研究し，必要に応じて改造する自由 (第一の自由)．ソースコードへのアクセスは，この前提条件となります．
> - 身近な人を助けられるよう，コピーを再配布する自由 (第二の自由)．
> - 改変した版を他に配布する自由 (第三の自由)．これにより，変更がコミュニティ全体にとって利益となる機会を提供できます．ソースコードへのアクセスは，この前提条件となります．
>
> 利用者が以上であげた全ての自由を有していれば，そのプログラムは自由ソフトウェアです．そうでなければ，不自由です．
>
> オープンソース?
>
> 別の人びとは，「オープンソース」という用語を「自由ソフトウェア」と近い (しかし同一ではない) ものを意味するのに使ます．わたしたちは「自由ソフトウェア」という用語のほうを好んでいます．(中略)「オープン」という言葉が自由に言及することはまったくありません..

コピーレフト (copyleft)

自由ソフトウェアの自由を法的に保護するため，GNU プロジェクトはコピーレフトを使っている．コピーレフトとは，すべての者が著作物を利用・再配布・改変できなければならない，という考え方．以下，GNU プロジェクトによる「コピーレフト」の定義 (`https://www.gnu.org/copyleft/copyleft.html`) の一部を引用する．

> コピーレフト (Copyleft) とは，プログラム (もしくはその他の著作物) を自由 (自由の意味において．「無償」ではなく) とし，加えてそのプログラムの改変ないし拡張されたバージョンもすべて自由であることを要求するための，一般的な手法の一つです．
>
> GNU プロジェクトにおいて，わたしたちの目的は，すべての利用者に対して GNU ソフトウェアを再配布し変更する自由を与えることです．もし，あいだに入った人がそういった自由を奪うことができるなら，わたしたちのコードがたとえ「多くの利用者を得る」ことができたとしても，そのコードは利用者に自由を与えないかもしれません．そこで，GNU ソフトウェアをパブリックドメインとせずに，わたしたちはそれに著作権 (コピーライト) ならぬ「コピーレフト」を主張することにしました．コピーレフトでは，そのソフトウェアを再配布する人は，変更してもしなくても，それをコピーし変更を加える自由を一緒に渡さなければならないということを主張します．コピーレフトによって，すべての利用者が自由を持つことが保証されるのです．
>
> コピーレフトは，プログラムの著作権を使う一つの方法です．著作権を放棄することを意味しません．実際，そうするとコピーレフトは不可能になってしまいます．「コピーレフト」の「レフト」は，動詞 “to leave” を指す言葉ではなく，「ライト」の鏡像の方向を指すだけの言葉です．
>
> 著作権のシンボルの代わりに，丸の中の逆方向の C を使うのは法的には間違いです．コピーレフトは法律的に著作権にもとづいています．ですから，作品には著作権表示があるべきです．著作権表示には，著作権のシンボル (丸の中に C) か，“Copyright” の言葉のどちらかが必要です．
> 丸の中の逆方向の C は，なんら特別な法的な重要性を持ちませんので，著作権表示となりません．それは，本の表紙や，ポスターなどでは楽しいでしょうが，ウェブページでどのように表現するかについては，注意を払いましょう!

第 II 部

計算機を使う

2　まずやってみる

何をやっているか理解しなくてもいいので，書いてある通りにやってみる．角のまるい枠で囲まれた部分は，読み飛ばしてよい．最初はとにかくやってみる．理解するのは後からでよい．

2.1　ログイン

ユーザーを選んでクリックする．薄い文字で “パスワード” と書かれた横長の窓が表示されるので，パスワードを入力する．入力した文字は全て ● で表示される．入力したら，『Enter』キーを押す．パスワードを正しく入力できていれば，しばらく待つと Ubuntu デスクトップが表示される (画面の左上の角に “Ubuntu デスクトップ” の表示が出る)．

2.2　ターミナル (端末) を立ち上げる

Ubuntu デスクトップで，『Ctrl』キーと『Alt』キーを押した状態で『T』のキーを押す．ターミナルウィンドウが開き，プロンプトが表示される．プロンプトの前には，ユーザー名やその他の情報も表示されているかもしれない．例えば，

```
george@astro:~$
```

以後，この文書ではプロンプトは全て「$」と書く．すなわち，実際には上のように表示されていても，

```
$
```

と書く．

プロンプト

プロンプトは，入力待ちの状態にあることを表わす記号である．プロンプトを表わす記号は $ と決まっているわけではない．例えば，% であったり，> であったり，あるいは # であったりする．設定を変えることによって，プロンプトは好きなものに変更することができるので，なんでもありだったりする．
いちおう，デフォルトのプロンプトはシェルによって決まっている．bash は $，csh は %，tcsh は > である．また，スーパーユーザ (root) のプロンプトは # である．

何かのキーを押しながら別のキーを押す

何かのキーを押しながら別のキーを押す操作は，「 + 」を使って表わすものとする．例えば，『A』のキーを押しながら『B』のキーを押す操作は，「A + B」と書く．『B』のキーを押しながら『A』のキーを押す場合は，「B + A」である．「A + B」と「B + A」は同じではない．ターミナルを立ち上げる操作は，「Ctrl + Alt + t」である．

Launcher に登録

画面の左端にはいくつかのアイコンが縦に並んでいる．ターミナルを立ち上げあげると，新しいアイコンが追加されて表示されるようになる．マウスを動かしてアイコンにカーソルを重ねると，吹き出しが出て情報が表示される．新しく追加されたアイコンにカーソルを重ねれば，“端末” と表示される．“端末” と表示された状態で，右クリックをすると，表示される項目が増える．右クリックして出てきた “Launcher に登録” にカーソルを合わせて “Launcher に登録” をハイライトし，クリックする．これをやっておくと，次回以降は最初からターミナルのアイコンが画面の左端に表示されるようになる．ターミナルのアイコンが表示されているときは，マウスをターミナルのアイコンに重ねて右クリック，“新しい端末” をハイライトしてクリック，という操作によってターミナルを起動することができる．

2.3 ターミナルに入力する

ターミナルウィンドウにコマンドを入力して計算機を操作する．キーボードからの入力をターミナルウィンドウに送るため，マウスのカーソルを入力したいウィンドウの上に移動し，ウィンドウをクリックする．クリックしたウィンドウでは，プロンプトの右にあるブロックが塗りつぶされる．以後，キーボードからの入力は，ブロックが塗りつぶされた

ウィンドウに送られる．

2.4 ディレクトリを作成する

ターミナルウィンドウを選択して，プロンプトの右にあるブロックが塗りつぶされた状態になったら，小文字で `mkdir work` とタイプする．画面は

```
$ mkdir work
```

となる．プロンプト `$` は入力前から表示されていたものであり，入力しない．また，大文字と小文字は異なるものと認識されるので，小文字で書くべきものを大文字で書いてはいけない．

ここで『Enter』キーを押すと，コマンド (この例であれば「`mkdir work`」) が実行される．画面には新しいプロンプトが現れて，

```
$ mkdir work
$
```

となる．

以下，上でおこなった一連の作業を

```
$ mkdir work
```

と書くことにする．行の先頭にある `$` はプロンプトである．`$` は入力しない (入力してはいけない)．また，最後に「Enter」キーを押すと書いていないが，入力したら最後に「Enter」キーを押す．

ディレクトリの作成

mkdir はディレクトリを作成する命令である．mkdir の後に 1 つ以上の半角スペースを挟んで文字列を書いて「Enter」キーを押すと，mkdir の後に書いた文字列の名前をもったディレクトリが作られる．

ディレクトリ

ディレクトリは住所のようなものだと思えばよい．1 カ所で全ての作業をおこなってもよいのだが，作業ごとに場所 (ディレクトリ) を分けることを強くお勧めする．

2.5 ディレクトリの確認

ディレクトリが作成されたことを確認する．

```
$ ls
```

上の作業をおこなうと，

```
$ ls
work
$
```

のように表示されるはずである．入力した行 (`$ ls`) と一番後ろの行 (`$`) の間には，work 以外にも複数の文字列が表示されているかもしれないが，それは問題ない．

文字列の中に work があることを確認する．work が文字列の中にない場合は，ディレクトリの作成に失敗しているので，「ディレクトリを作成する」に戻って作業をやり直す．

ファイル一覧の表示

`ls` はファイルの一覧を表示する命令である．`ls` の後に 1 つ以上の半角スペースを挟んで文字列を書いて「Enter」キーを押すと，`ls` の後に書いた文字列の名前をもったディレクトリにあるファイルの一覧が表示される．文字列を省略して `ls` とだけ入力すると，カレントディレクトリにあるファイルの一覧を表示する．

カレントディレクトリ

現在地をカレントディレクトリと呼ぶ．

2.6 ディレクトリの移動

作成したディレクトリに移動する．

```
$ cd work
```

ディレクトリの移動

`cd` はディレクトリを移動する命令である．`cd` の後に 1 つ以上の半角スペースを挟んで文字列を書いて「Enter」キーを押すと，`cd` の後に書いた文字列の名前をもったディレクトリに移動する．文字列を省略して `cd` とだけ入力すると，ホームディレクトリに移動する．

ホームディレクトリ

ホームディレクトリはユーザーが好きなように使うことのできる場所である．ユーザーの自宅をホームディレクトリと呼んでいるのだと思ったらよい．

2.7 カレントディレクトリの確認

現在地を確認する．

```
$ pwd
```

上のように入力した後の結果は，ユーザー名によって変わってくる．例えば，ユーザー名が george である場合には，

```
/home/george/work
```

となる．ユーザー名が hogehoge の場合には，

```
/home/hogehoge/work
```

である．

各自のユーザー名に対応したものが表示されなかった場合には，

```
$ cd
```

とした後，「ディレクトリの移動」に戻って作業をやり直す．

カレントディレクトリの確認

pwd はカレントディレクトリを表示する命令である．

ディレクトリの読み方

ディレクトリは日本の住所のような構造になっている．例えば，岡山県岡山市北区津島中という住所は，岡山県下の岡山市下の北区下の津島中，を表わしている．同様に，/home/george/work は，/ の下の home の下の george の下の work，である．いちばん大きな領域は / で表わされ，これは root と呼ばれる．先頭の / は root を表わしているが，それ以外の場所に出てくる / は切れ目を表わす記号で意味をもたない．

2.8 資源の入手

サンプル・コードを入手する．

```
$ cp /home/george/student/fortran/samples.tar .
$ ls
```

これは，

```
$ cp /home/george/student/fortran/samples.tar .
```

と

```
$ ls
```

を続けておこなう，という意味である．

画面は

```
$ cp /home/george/student/fortran/samples.tar .
$ ls
```

```
samples.tar
$
```

のようになる．samples.tar が表示されない場合には，「資源の入手」の最初に戻って作業をやり直す．

コピー

cp はファイルの複製をつくる命令である．cp の後に 1 つ以上の半角スペースを挟み，1 つ以上の半角スペースを挟んで 2 つの文字列を書く．2 つの文字列のうち最初の文字列で表わされるファイルの複製が，2 つ目の文字列の名前をもったファイルとして作られる．2 つ目の文字列に「.」を指定すると，コピー元のファイルと同じファイル名をもったファイルがカレントディレクトリに作られる．

ファイル

ファイルとは，ラベル (ファイル名) の貼られたノートのようなもの，であると思ったらよい．ファイルは，ディレクトリとファイル名によって特定される．例えば，
/home/george/student/fortran/samples.tar
は，/home/george/student/fortran というディレクトリの下にある，samples.tar という名前がつけられたファイル (“samples.tar” というラベルの貼られたノート) である．ひとつの場所 (ディレクトリ) に同じ名前を持ったファイルが複数存在することはできない．場所 (ディレクトリ) が異なれば，同一のファイル名を持ったファイルが存在してもよい．すなわち，以下の 3 つ
/home/george/student/fortran/samples.tar
/home/george/student/python/samples.tar
/home/george/student/samples.tar
は，全て別のファイルを表わす．ファイル名とファイルの中身は対応していなくてもよい[a]．上の 3 つのファイルは同じ名前だが，中身は同じであっても同じでなくてもよい．

[a] 現実の世界でもラベルと中身が一致しないことはよくある．一致しない原因は，ずぼらなだけという場合もあるが，意図的におこなわれる場合もある．例えば，秘密の日記に「量子力学・講義ノート」というラベルを貼ってカモフラージュする．

2.9 展開

ファイルを展開する．

```
$ tar xvf samples.tar
```

結果の確認．

```
$ ls
samples  samples.tar
```

1 行目は入力を表わす．2 行目は計算機が表示したもので，入力ではない (行の先頭に $

がない行は計算機が表示したものである)．計算機が表示した文字列の中に samples があることを確認する．表示された文字列の中に samples がない場合には，「解凍」の最初に戻って作業をやり直す．

展開

複数のファイルをまとめたファイルを，元のバラバラの状態に戻すことを「展開する」と言う．
tar はファイルを展開したり，その反対をしたり (複数のファイルをまとめたり) する命令である．展開するのかその反対をするのかは，tar の後に書く文字列で指定する．動作を制御する文字列は，オプションと呼ばれる．

2.10 移動

移動．

```
$ cd samples/001
```

確認．

```
$ pwd
/home/george/work/samples/001
```

george を各自のユーザー名で置き換えた文字列が表示されていることを確認する．異なる文字列が表示された場合には，

```
$ cd
$ cd work
```

としてから，「移動」の最初に戻って作業をやり直す．

2.11 コンパイル

コードを機械語に翻訳する.

```
$ gfortran test001.f
```

確認する.

```
$ ls
a.out  test001.f
```

a.out があることを確認する.

コンパイル

プログラミング言語によって記述されたコードを機械語に翻訳することを,「コンパイル」と言う. 計算機が理解するのは機械語である. 人間がプログラミング言語を使って書いたコードを計算機に実行させるためには, コードを機械語に翻訳する必要がある.

gfortran は, Fortran というプログラミング言語で記述されたコードを機械語に翻訳する命令. gfortran の後に 1 つ以上の半角スペースを挟んで文字列を書いて『Enter』キーを押すと, 文字列で指定されたファイルの中身 (コード) が, 機械語に翻訳される. 機械語に翻訳されたものは, カレントディレクトリに a.out という名前のファイルとして保存される.

スクリプト言語とコンパイラ言語

プログラミング言語にはスクリプト言語とコンパイラ言語がある．
スクリプト言語で書かれたコードは，翻訳と命令の実行が同時におこなわれる．スクリプト言語で書かれたコードの命令は，コードを指定するだけですぐに実行することができる．スクリプト言語には，Python, Ruby などがある．
コンパイラ言語で書かれたコードは，翻訳と命令の実行を別々におこなう．コンパイラ言語で書かれたコードを実行するためには，まずコンパイル (翻訳) の作業をおこなって，機械語で記述されたプログラム (ファイル) を作成する．それから，プログラムを指定して実行する．コンパイラ言語はコードをコンパイルするときに時間をかけて最適化をおこなうため，動作はスクリプト言語よりも速くなる．コンパイラ言語には，Frotran, C などがある．

2.12 実行

実行する．

```
$ ./a.out
Hello World
```

Hello World と表示されれば成功．

プログラムの実行

プログラムを実行するには，ファイル名を入力する．例えば，/home/george/work/samples の下にある a.out という名前のファイルを実行するには，
$ /home/george/work/samples/a.out
とする．

絶対パスと相対パス

ファイルはディレクトリとファイル名を使って指定するが，指定の方法には絶対パスと相対パスの 2 通りの方法がある．/home/george/work/samples/a.out のように，root から始めて全てのディレクトリを書き下してファイルを指定する方法は，「絶対パス」と呼ばれる．もうひとつ，カレントディレクトリを基準としてファイル名を指定する「相対パス」と呼ばれる方法がある．カレントディレクトリは，「.」で表わすことができる．すなわち，./a.out はカレントディレクトリにある a.out という名前のファイルを指す．

2.13 第一段階修了

ここまで辿り着いたということは，第一段階を修了したということである．何をやったのかわかっていないかもしれないが，最初はそんなものである．これからひとつずつ理解していったらよい．

3　コードを書いてみる

前節 (まずやってみる) では，サンプルのコードをコンパイルして，実行した．この節では，自分でコードを書いてみて，自分で書いたコードをコンパイル，実行する．前節と同様に，まずはやってみる．理解するのは後からでよい．

3.1　ファイル名

コードを保存したファイルはソースファイルと呼ばれる．Fortran のソースファイルの拡張子は `.f` でなければならない[25]．

拡張子 (extension)

ファイルの種類を判別するために，ファイル名の末尾にピリオドに続いて付される 3–4 文字程度の文字列．拡張子の多くは慣習的に利用されているもので，例えば，テキストファイルには `txt`，JPEG ファイルには `jpg`，PDF ファイルには `pdf`，などが使われる．
拡張子はファイルの中身と関係なく付けることができる．拡張子でファイルの中身が確定する (拡張子を見たらファイルの種類を特定できる) というものではない[a]．逆に，拡張子を変更してもファイルの中身が書き変わるわけではない[b]．

[a] 普通はファイルの中身と拡張子を一致させる (わざわざ紛らわしくなるようなことはしない) が，悪意のある攻撃者は拡張子の変更を偽装のためにおこなうことがある．例えば，悪いプログラム (実行型ファイル) に hentai.jpg といった名前をつけて画像ファイルに偽装する．考えなしの人が JPEG の画像ファイルだと思ってクリックすると，悪意のあるプログラムが実行されてしまう．

[b] 「Exel のファイルは使い勝手が悪いから，csv に変換して送ってください」と言ったら，ファイル名を hogehoge.xls から hogehoge.csv に変更しただけのファイルが送られてきたことがあった．ファイル名 (拡張子) を xls から csv に変更しても，中身が csv になったりはしない．

3.2　Emacs を使う

コードを書くには，エディタと呼ばれる種類のソフトウェアを使う．ここでは Emacs と呼ばれるエディタを使ってみる．好みのエディタがあるなら，それを使ってもらって

[25] 処理系によっては `.for` とするものもあったかもしれない．

よい．

Emacs を起動する．

```
$ emacs -nw sphere.f
```

正しく Emacs が起動すれば，ターミナルウィンドウの見た目が変わるはずである．上のようにして起動すると，sphere.f という名前のファイルを編集できる状態になる．キーボードを叩けば，叩いたキーの文字がウィンドウに表示される．『Enter』キーを叩くと，改行される．

ここでは，以下のコード (コード 1) を入力する．これは球の表面積と体積を計算するコードである．とりあえず，内容を理解する必要はない．まずは，エディタを使ってファイルを作成できるようになることが重要である．

Emacs でよく使うコマンドを表 1 にまとめた．この表に載せた以外にも便利なコマンドがたくさんあるので，気になる人は調べてみるとよい．入力ミスした文字を消去するには「Ctrl + d」．これは，『Ctrl』キーを押しながら『D』のキーを押す，である．ファイルを保存するときは「Ctrl + x，Ctrl + s」．これは，『Ctrl』キーを押しながら『X』のキーを押し，さらに続けて『Ctrl』キーを押しながら『S』を押す，である．

あと，空白にも意味がある場合があるので，コードの入力では空白もきちんと入力する必要がある．空白の数にも意味がある場合があるので，きちんと数を数えて入力する．例えば，1 行目にある “program” の前に入る空白は半角で 6 個である．半角の空白 2 個と全角の空白 1 個は同じではない．半角の空白は，半角の空白として入力しなければならない．

コード 1　sphere.f

```
1       program sphere
2       implicit none
3 c constant
4       real PI
5       parameter (PI=3.141593)
6 c local variable
7       real radius
8       real area, volume
9 c begin
```

```
10        radius = 10.0
11        area   = 4.0 * PI * radius**2
12        volume = 4.0 / 3.0 * PI * radius**3
13        write(*,*) 'radius:', radius
14        write(*,*) '  surface area =', area
15        write(*,*) '  volume       =', volume
16  c end
17        stop
18        end
```

表 1　Emacs でよく使うコマンド

カーソルを右の文字へ	Ctrl + f
カーソルを左の文字へ	Ctrl + b
カーソルを上の文字へ	Ctrl + p
カーソルを下の文字へ	Ctrl + n
カーソルの前の位置にある文字の消去	Ctrl + h
カーソルの位置にある文字の消去	Ctrl + d
ファイルの保存	Ctrl + x, Ctrl + s
終了	Ctrl + x, Ctrl + c

3.3　コンパイルして実行する

コードの入力が終わって，ファイルに無事保存されたら，コンパイルする．コードを保存したファイルがあるディレクトリに移動して，

```
$ gfortran sphere.f
```

とやってみる．ソースコードのファイル名は `.f` で終わっていなければならない (3.1 節)．正しくファイルが作られていれば，特にメッセージを吐き出すことなく，新しいプロンプトが表示されるだろう．

不幸にしてエラーメッセージが吐き出された場合は，コードの入力に失敗しているの

で，エディタを使ってファイル sphere.f を修正する．エラーメッセージをよく読むとどこに間違いがあるかわかる場合もあるが，初心者には何がなんだかわからないと思う．幸いにして sphere.f はそれほど長いコードではないので，初心者は愚直にコードを頭から見直して間違いを探すのが，コードを修正する近道と思う．間違いを直したらファイルに保存して，コンパイルする．

```
$ gfortran sphere.f
```

コンパイルできるまで，コードの修正をくり返す．

正しくコンパイルされた場合は，ファイルの一覧を表示させると

```
$ ls
a.out  sphere.f
```

となる．a.out は sphere.f が機械語に翻訳されたプログラムである．実行するときは，ファイル名を指定する．

```
$ ./a.out
```

コードが正しく入力されていれば，以下のように，球の半径，表面積，体積，が表示される．

```
 radius: 10.
   surface area = 1256.6372
   volume       = 4188.791
```

第 III 部

Fortran

4　FORTRAN 77 の簡単な解説

4.1　FORTRAN 77 の約束事

`sphere.f` は，FORTRAN 77 と呼ばれる 1977 年に制定された規格に従って書かれたコードである．カードでプログラムを入力していた時代から使われていたせいで，コードはいくつかの摩訶不思議な約束事に従って書かれている．

- プログラムを構成する文は，1 行の 7 文字目から 72 文字目までに書く
  - 多くの行で最初の 6 文字が空白になっているのはこのため
  - 6 文字目に `&` などの文字を入れると，その行は前の行の続きとして処理される
  - 6 文字目に文字を入れることで，3 行でも 4 行でも好きなだけ伸ばすことができる
- 1 文字目に `c` あるいは `*` のある行はコンパイル時に無視される
  - コンパイル時に無視されるものを「コメント」と呼ぶ
  - コメントは 7 文字目から 72 文字目以外にも書き込み可
- 1 文字目から 5 文字目までの数字列は行番号を表わす
- 大文字と小文字を区別しない[26]
- プログラムは program 文で名前を宣言し，stop 文と end 文で終わる

こんなことを並べられても，何がなんだかわからないのが普通だろう．まずは，意味がわからないままに作成した `sphere.f` を解読するところから始めてみる．

[26] FORTRAN77 は全て大文字で書くことになっているが，FORTRAN77 のコンパイラは小文字を大文字に置き換えてコンパイルするので，コードを小文字で書いても問題はない．コードを大文字で書くか小文字で書くのかは，どっちを使ったらコードが読みやすくなるかという観点で決めたらよい．コードには大文字と小文字が混ざっていても問題ないので，積極的に大文字と小文字を混ぜてコードを書くという方法もある．例えば，コードの中で値が変わる変数は小文字，値が変わらない変数 (例えば `parameter` 文で値を設定する変数) は大文字，などといったルールに従ってコードを書くと，後でコードを読み返すのが楽になることがあるかもしれない，ただし，コンパイル時に小文字は大文字に変換されてしまうので，コードの中で変数 `A` と変数 `a` を同時に使うことはできない．変数 `A` と変数 `a` の両方が使われているコードをコンパイルすると，変数 `a` は変数 `A` に置き換えられ，変数 `A` だけの (変数 `a` を含まない) プログラムが生成する．

4.2 sphere.f を解読する

sphere.f を再掲する.

コード 2 sphere.f

```
1         program sphere
2         implicit none
3   c constant
4         real PI
5         parameter (PI=3.141593)
6   c local variable
7         real radius
8         real area, volume
9   c begin
10        radius = 10.0
11        area   = 4.0 * PI * radius**2
12        volume = 4.0 / 3.0 * PI * radius**3
13        write(*,*) 'radius:', radius
14        write(*,*) '  surface area =', area
15        write(*,*) '  volume       =', volume
16  c end
17        stop
18        end
```

コードは基本的に前から順番に実行される.

1 行目には, program に続けてプログラムの名前を書く. 名前はだいたいなんでもよいが, Fortran の予約語 (program などのキーワード) は使えない. FORTRAN 77 の約束事に従って, 最初の 6 文字は空白になっている.

2 行目はおまじない. このおまじないの意味は「暗黙の型宣言を禁止する」である. 暗黙の型宣言というのは, 宣言されていない変数を使ったとき, 変数の名前に応じて自動的に変数の宣言がなされるという機能である (「変数の宣言」については 4 行目で説明する). 暗黙の型宣言はバグの元になるので, 使わないことを強く推奨する. 暗黙の型宣言はデフォルトで有効になっているので, implicit none のおまじないを唱えて無効化する.

3 行目はコメント．コメントはコンパイル時に無視されるものであり，コメントはプログラムの動作に全く影響を及ぼさない．すなわち，コメントはプログラムとしては意味をもたない．意味をもたないものをわざわざ書くのは，コードの可読性を高めるためである．コードはプログラミング言語の規約に従って書かなければならないため，必ずしも人間にとって理解しやすいものとはならない[27]．自分が書いたコードでも，3 日経つと解読不可能な暗号[28]になる，というのはよくあることである．コメントは，コードが解読不可能な暗号になることを防ぐために書くのである[29]．解読できないコードは怖くて使えない．コードの可読性を高めるため，コメントを積極的に活用するべきである[30]．

4 行目．変数の宣言．変数の宣言とは，ある文字列をこれこれのデータ型の変数として使うと決めること，である．4 行目は，`PI` という文字列を実数型の変数として使う，という意味である．Fortran で使うことのできるデータ型には，実数型，整数型，文字型，などがある．データ型の詳細な説明はまた後でおこなう．とりあえず，このコードにおいて `PI` は実数である，ということだけわかればよい．`implicit none` のおまじないを唱えているなら，コード内で使用する変数を全て明示的に宣言しなければならない．宣言していない変数を使うと，コンパイル・エラーとなる．

5 行目では，変数 `PI` に値を代入している．`parameter` を使って値を代入された変数は定数となり，値を変更できなくなる．Fortran では，「左辺=右辺」で，右辺の値を左辺の変数の値とする．左辺は変数でなければならない．右辺は数値，または数式．数式はその段階で数値計算可能でなければならない．

6 行目はコメント．これに続く行では変数の宣言がおこなわれるのだろうと，予測される．

7 行目と 8 行目は変数の宣言 `radius`，`area`，`volume` の 3 つの実数型の変数が宣言されている．

9 行目はコメント．ここからプログラムの本体が始まるのだと予測される．

10 行目．変数 `radius` に値を代入している．

11 行目．右辺で表面積の計算をして，結果を左辺の変数 `area` に代入する．`**` は冪乗

---

[27] ウィザード (wizard) 級になると，自然言語よりプログラミング言語の方がよくわかる，という人もいるらしい．

[28] そもそも「コード (code)」というのは暗号という意味である

[29] コメントは未来の自分のために書くものである．3 日後の自分は他人なので，コメントは他人が読んで理解できるものでないと役に立たない．自分ならわかるだろうと思って書いたコメントは，未来の自分の役に立たない可能性が非常に高い．3 日後の自分はとんでもなく間抜けだと思っておいた方がよい．

[30] 大学院生だったときに聞いた話．とある大学で気象庁からもらったコードをひも解いてみたところ，コメントがひとつも書かれていなかった．コードをくれた人に問い合わせしたところ，不要だと思ってコメントは全て消したとの返事．コメントがないコードの解読は困難を極めた．あまりにも辛い作業に結局は解読を断念し，コードをゼロから書くことになった．

をあらわす．Fortran で `x**y` は x^y である．

12 行目．体積の計算．

13–15 行目．結果の出力．`write(*,*)` は結果を出力する命令．`(*,*)` の部分を書き換えることでいろんなことができるのだが，とりあえず，こう書くものだと思っておいたらよい．クォーテーション (') やダブルクォーテーション (") で括った部分は，そのまま出力される．括られていないものは変数と認識され，変数に格納されているデータが出力される．

16 行目．コメント．

17 行目．プログラムを終了する合図．なくてもよい場合も多いのだが，書いておくのが無難である．

18 行目．`end` は 1 行目の `program` と対になり，`program` と `end` に囲まれた部分がプログラムであることを示す．

円周率

円周率を小数点以下 2 桁まで覚えていることは常識であって欲しいと思うが，それ以上の桁まで覚えている人[a]は稀だと思う．上にあげた `sphere.f` では小数点以下 6 桁までの数字を書いているが，そこまで覚えていないという人は，組み込み関数[b]を使うことで円周率の数値を設定することができる．例えば，`arccos` を使うなら，`PI=acos(-1.0)` とすればよい．

[a] 世間では「変態」と呼ばれる．

[b] Fortran の組み込み関数は 6.2.3 節を参照のこと．

5　プログラミング入門

5.1　基本部品

Fortran を使って数値計算するときの基本部品は，

- 変数の宣言
- 計算
- 条件分岐 (繰り返しを含む)
- 入出力

に分けられる．

まず，４つの基本部品を使ったコードを解説し，基本部品それぞれについての解説は次節でおこなう．

5.2　基本部品を使ったコード

次のコード `sum1.f` は級数

$$\sum_{k=1}^{n} k = 1 + 2 + \cdots + n \tag{1}$$

を計算する．

コード 3　sum1.f

```
1         program sum1
2         implicit none
3   c -----
4         integer n, ns, k
5   c -----
6   c input 'n'
7    1    continue
8         write(*,*) 'input n'
9         read(*,*) n
10        if (n.le.0) then
11            go to 1
12        end if
```

```
13 c calculation
14       ns = 0
15       do 100 k = 1, n
16         ns = ns + k
17  100  continue
18 c output
19       write(*,*) '1+2+...+n ='
20       write(*,*) ns
21 c -----
22       stop
23       end
```

これもまた FORTRAN 77 で書かれている．

1 行目は program に続けてプログラムの名前を書く．

2 行目は「暗黙の型宣言を禁止する」おまじない．

3 行目は先頭に c とあるのでコメント行．コードの可読性を高めるためだけに書かれた行である[31]．

4 行目．整数型の変数 n, ns, i を宣言．

5 行目．コメント行．変数の宣言が終わり，ここから処理に関する記述がはじまる．

6 行目．プログラムの動作を説明するコメント．これに続く 7–12 行目では，式 1 の n に入れる値をキーボードから読み込む．

7–12 行目はひとつの固まり．

7 行目の continue は，何もしない，という命令．行番号 1．

8 行目は，input n と出力する．

9 行目は，入力された文字を読んで，読んだ値を変数 n に代入する．例えば，n に 3 を代入するときは，キーボードの『3』のキーを叩いた後，『Enter』キーを叩く．

10–12 行目は，ひとつの固まり．If 文と呼ばれる条件分岐のひとつ，である．括弧の中が真のとき，if と end if の間が実行される．括弧の中が偽のときは何もせず，end if の次の行にとぶ．括弧の中の n.le.0 は，$n \leq 0$ を意味する．すなわち，変数 n が 0 か負の場合に真となり，変数 n が正の場合は偽となる．したがって，変数 n が 0 か負の場合に 11 行目が実行され，それ以外の場合には何もせず 14 行目に移動する．

11 行目の go to は，命令を実行する順番を変更する命令．プログラムは前から順番に

[31] こんなもので可読性が高まるとは思わない，という人もいるかもしれない．人によって何を良いと考えるのかは違うので，それぞれ自分の好みに合ったスタイルを作ってもらったらよい．

実行され，ひとつ実行するごとに次の行へと移動していくことが基本であるが，go to 文の次は，go to の後に指定した行番号を実行する．

10–12 行目は，変数 n が正の値のときは何もしないが，変数 n が 0 または負の値のときには 11 行目が実行され，次には行番号 1 (コードの 9 行目) が実行される．9 行目は，キーボードから入力された値を変数 n に代入する．すなわち，変数 n に 0 または負の値が代入されたときには，変数 n の入力をやり直させるようになっている．

13 行目のコメントは，14–17 行目で実行することの説明である．14–17 行目では，級数の計算がおこなわれる．

14 行目は，変数 ns に 0 を代入している．変数は使う前に必ず値を代入する．Fortran では，= を使うと右辺の変数に左辺の値が代入される．

15–17 行目はひとつの固まり．do 文は繰り返し処理の命令である (繰り返しもまた条件分岐のひとつ)．do 文は，do の後に行番号を書いて，繰り返しの範囲を指定する．このコードでは行番号として 100 が指定されているので，do の行 (15 行目) と行番号 100 の行 (17 行目) 間にある行 (この例だと 16 行目) が，繰り返し実行される．do 文の行番号の後ろ (このコードでは，k = 1，n と書かれた部分) は，繰り返し回数を指定する．k は do 文を制御する変数で，この変数は必ず整数型を使わなければならない．do 文を制御する変数には，まず = の右に書かれた 2 つの数値のうち左に書かれた数値 (このコードでは 1) が代入される．次に，do 文の内側 (行番号で指定した繰り返し範囲) を 1 回実行する．実行し終えたら，制御変数の値を 1 増やす．増やした後の数値を，= の右に書かれた 2 つの数値のうち右に書かれた数値 (このコードでは n) と比較し，制御変数の方が右に書かれた数値と同じかそれ以下であれば，do 文の内側をもう 1 回実行する．制御変数の方が大きいなら，繰り返しを終了する (このコードであれば 18 行目に移動する)．

16 行目は，変数 ns に値を代入している．まず最初は，ns=0 かつ k=1 なので，右辺は 0+1 であり，ns には 1 が代入される．繰り返し範囲の実行が終了したので，制御変数の値は 1 増えて k=2 になる．n が 1 であるなら，制御変数の値はそれより大きいので，do 文はこれで完了となる．n が 2 かそれよりも大きい場合は，do 文の内側が再び実行される．今度は，ns=1 かつ k=2 なので，ns には 3 が代入される．繰り返し範囲の実行が終了したら，制御変数の値は 1 増えて k=3 になる．n が 2 であるなら，制御変数の値はそれより大きいので，do 文はこれで完了となる．n が 3 かそれよりも大きい場合は，do 文の内側が再び実行される．今度は，ns=3 かつ k=3 なので，ns には 6 が代入される．繰り返し範囲の実行が終了したら，制御変数の値は 1 増える．繰り返しが終りになったかどうか判定する．この繰り返しである．

18 行目，コメント行．結果の出力に関する記述がはじまる．

19 行目はクォーテーション (') で括られた文字列 `1+2+...+n =` がそのまま出力される.

20 行目は変数 `ns` の値を出力. `write` 文は, ' で括られていない文字列を変数と解釈し, 変数に代入されている数値や文字列を出力される.

21 行目, コメント行.

22 行目はプログラムを終了する合図.

23 行目は, program と対になってプログラムの範囲を表わす.

字下げ

コード `sum1.f` では, 11 行目と 16 行目で字下げと呼ばれるテクニックが使われている. 字下げは, プログラムの構造を見やすくするために使われる.
このコードの例であれば, 11 行目は `if` 文が真であるときにのみ実行され, 偽であるときには実行されない. `if` 文に対して字下げされることで, この行は `if` 文によって実行が制御されているのだとわかる. 同様に, 16 行目は `do` 文で実行が制御される部分である.
字下げを適切におこなうことでコードの可読性は大いに高まる. 可読性を高めることは, バグを減らすことにもつながる. 一貫した適切なルールに基づいて, 積極的に字下げをおこなうべきである.

コードを解読したら, 実際に動かしてみる. コンパイルは

```
$ gfortran sum1.f
```

実行は

```
$ ./a.out
```

ちゃんとできていれば

```
input n
```

と表示されて，式 1 の n の値の入力を待つ状態になる (ここでプロンプト ($) は表示されない)．キーボードを使って適当な数値 (整数) を入力する (最後に『Enter』キーを押す)．例えば 10 を入力すると，

```
10
1+2+...+n =
55
$
```

となって，プログラムは計算結果を表示して終了する (プログラムが終了するとプロンプト ($) が表示される)．ちなみに，

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2} \tag{2}$$

である．プログラムが表示した計算結果が正しいことを確認する．

コードが正しく動作していることを確認するため，違う数値を入力して，正しい結果が出ることを確認する．このコードは，0 以下の数値が入力されたときには，入力のやり直しを求めるようになっている．例えば，

```
input n
-3
input n
```

0 以下の数値を入力して，期待通りに動作することを確認する．

このコードは，read 文が読み込んだ値を整数型の変数 n に代入するようになっていて，整数以外の値が入力されることを想定していない悪いコードである[32]．整数以外のも

[32] ちゃんとしたコードは，想定外の入力ににも対応できるように書かれているものである．しかし，あらゆ

の (実数とか文字とか) を入力して，何が起こるか見てみるのもいいかもしれない．

5.3 デバッグの方法

コードに含まれる誤りはバグ (bug)[33]と呼ばれる．バグを取り除く作業をデバッグ (debug) と言う．

ある種のバグは，コンパイル時にエラーを出して，コンパイルを失敗させる．そういう種類のバグは，コンパイル時に吐き出されるエラーメッセージを解読することで，比較的簡単に場所を特定して修正することができる．

一方で，コンパイル時に問題を生じさせないバグがある．コンパイルはできるのだが，実行してみると期待したように動作しないため，問題があるのだとわかる．明らかに動作が期待通りではないとわかる場合はまだよい．一見したところ正常に動作しているように見えて実は間違っている，ということはよくあるのだが，このタイプのバグを見つけるのは難しい．コードを書いてみたら，答えがわかっている場合について実行してみて，プログラムが期待通りに動作していることを確認することが重要である．答えがわからないからコードを書いて計算しようというのに，答えがわかっているものを計算するというのはヘンテコな話であるように思われるかもしれないが，答えがわかっているものを計算することはめちゃくちゃ重要なことなのである．

デバッグの方法はいろいろあるが，基本となるのは `write` 文を使う方法である．コードのあちこちに `write` 文を埋め込んで，いろいろな変数を出力させて，プログラムが意図した通りに動作しているかどうかを確認する．変数に代入されている数値が意図したものになっていないなら，それより前で何かを間違えているのだとわかる．`write` 文の位置を変えることでバグを特定し，バグを修正する．

デバッグの作業においては，デバッガと呼ばれる種類のソフトウェアを活用することも

ることを想定してコードを書くのは面倒なので，手を抜いて想定外の入力に対応しないコードを書くことは，よくあることである．そして，そのせいで痛い目に遭ったことは何度もあるのだが，それでもやっぱり手を抜いてしまうことが多い．そもそも想定外はめったに起こらない (頻繁に発生するならそれは想定内でなければならない)．なので，手を抜いても問題にならないことは多い (顕在化しないものは問題となりえない)．めったに起こらないことに労力を割くのはコストパフォーマンスが悪い．コードを書くときに想定外をどこまで考えるのか，そのさじ加減は難しい．

[33] あるとき，電気機械式計算機「Mark II」が停止して，その原因を探ると機械の中に蛾が挟まっているのが見つかった．グレース・マレー・ホッパー (Grace Murray Hopper, 1906–1992) は，この蛾を作業日誌に貼り付けて，”First actual case of bug being found“と書き込んだ．このエピソードをきっかけに「バグ」という言葉がコンピュータ・プログラム上の誤りや欠陥を表わすものとして広まったとされている．先駆的な計算機科学者であるホッパーは，世界初のコンパイラ A-0 System の開発者であり，プログラミング言語 COBOL の開発者でもある．また，彼女はアメリカ海軍軍人 (最終階級は准将) でもあり，アメリカ海軍のミサイル駆逐艦ホッパー (USS Hopper) の艦名は彼女に因んで命名された．

できる．デバッガにもいろいろあるので，興味をもった人は調べてみるとよい．

最後にもうひとつ，たぶん大事なこと．期待通りに動作しない原因はバグのせいとは限らない．実装したアルゴリズムに誤りがあれば，期待した通りに動作しない．当たり前であるが，なかなかやっかいな問題であったりする．どこに間違いがあるのかを見つけるのは，難しいのである．

計算機は融通が利かない

コードを書いてみたらすぐにわかると思うが，計算機は全く融通が利かない[a]．コードは規約に従って正しく書かれていなければならない．空白も規約に従って機械的[b]に解釈される．空白があってはいけないところに空白があってはならないし，空白がなければならない場所には空白がなければならない．人は少しくらい間違っていても自動的に修正して理解してくれるが，計算機は全く融通が利かない．

しかしながら，融通が利かないのは悪いことばかりではない．計算機は余計なことを絶対にしない．これは素晴らしい性質である．人にものを頼むと，やる必要のない余計なことをやってくれたせいで全てが台無しになる，ということがあったりする．計算機はコードに書いたことだけを，コードに書かれた通りに実行する．この安心感は，計算機を使う大きな理由のひとつである．

[a] “コンピュータは期待している通りではなく，プログラミングされた通りに動く．” (清水亮『教養としてのプログラミング講座』中公新書ラクレ)

[b] 計算機は機械なのだから，定義によって「機械的」以外の動作はありえないのだが．

ベア・プログラミング

プログラミング作法という超有名な本 [3] には，以下のような記述がある．

> 自分のコードを他人に説明してみよう．自分のコードを誰かほかの人に説明して聞かせるのも効果的なテクニックだ．こうすると自分自身インバグが見えてくることが多い．場合によっては説明し始めた途端に気がついて「あ，もういいや，変なところがわかったよ．ごめん，ごめん」などと言って照れくさい思いをすることもある．このテクニックは意外なほど有効だし，聞き手は別にプログラマでなくてもかまわない．
>
> ある大学の計算機センターのヘルプデスクのそばにはテディベアのぬいぐるみが常備されており，摩訶不思議なバグに悩む学生は，人間のスタッフに相談する前にぬいぐるみに向かって説明しなければならないことになっていた．

「意外なほど有効」という言葉に間違いはない．こんなのでうまくいくはずがないと思うかもしれないが，これは本当にうまくいくのである．本当に本当なのである[a]．

[a] 「本当だ」と言えば言うほど嘘っぽくなるのはなぜなのだろうか？

6　基本部品の解説

6.1　変数の宣言

6.1.1　変数の型

Fortran が扱う変数には，整数，実数，複素数，論理型，文字型，がある．とりあえず，整数型と実数型がわかればよい．余裕があれば，文字型も使いこなせるようになっておくと便利ではある[34]．

表 2　Fortran で扱われる変数の型

型	Fortran	大きさ
整数型	`integer`	$-32768 \sim 32767$
	`integer*4`	$-2147483648 \sim 2147487647$
実数型	`real`	$\pm 0.29 \times 10^{-38} \sim \pm 1.7 \times 10^{38}$ (有効数字 7 桁)
	`real*8`	$\pm 0.29 \times 10^{-38} \sim \pm 1.7 \times 10^{38}$ (有効数字 15 桁)
文字型	`character*`n	n の長さの文字列 (例：`character*12`)

変数名は基本的に英数字とアンダースコア (_) を使って最大 31 文字まで[35]で指定する．ただし，変数名は英字で始まるものでなければならない．また，変数として使うことのできない文字列がいくつかある．例えば，組み込み関数として使われている `sin`, `log`, `exp`, などは変数名として使うことはできない．一方で，`do` や `if` は Fortran で意味を持つ文字列であるが，変数名として使うことが可能である．可能ではあるが，そんなものを変数名として使用するとコードの可読性が著しく損なわれるから，普通の人は使ったりしないだろう[36]．

6.1.2　宣言文

変数は以下のように宣言する．

---

[34] Fortran で文字列の操作をするのは面倒なので，文字列を扱う処理は Fortran ではない他の言語を使っておこなうことが多い．

[35] Fortran 90 の標準規格の場合．FORTRAN 77 は最大 8 文字まで．

[36] 嫌がらせを目的にしてコードを書くというあまりありそうにない状況においては，`do` や `if` を変数名として使うということがあるかもしれない．妄想はできるが，嫌がらせでコードを書くという状況は，やっぱりありそうな気がしない．

コード 4　変数の宣言 1

```
1       integer       i
2       real          radius
3       real          area , volume
4       character*8   char
```

1 行目は，i という変数が整数型であることを宣言している．2 行目と 3 行目は，radius，area, volume という 3 つの変数が実数型であることを宣言している．3 行目のように同じ型の変数は，カンマ (,) で区切って並べればまとめて宣言することもできる．4 行目は，char という変数が 8 文字からなる文字列であることを宣言している．8 の代わりに 12 を用いれば，12 文字からなる文字列になる．

x_i $(i = 1, 2, ..., 10)$ を変数として使いたいときは，以下のように宣言する．

コード 5　変数の宣言 2

```
1       real   x(10)
```

こうすると，x(1), x(2)，..., x(10) を使うことができるようになる．2 次元や 3 次元の変数も可能である．例えば，

コード 6　変数の宣言 3

```
1       real   y(3,5)
2       real   z(10,20,30)
```

要素の数は変数を使って指定することもできる．

コード 7　変数の宣言 4

```
1       integer i,j
2       parameter (i=3)
3       parameter (j=5)
4       real   x(i)
5       real   y(i,j)
```

要素の大きさを指定する i と j は，それより前で parameter 文で値を設定しておく．FORTRAN 77 では，上の例の 2 行目や 3 行目がない形での変数の宣言は認められていない[37]

[37] 要素の数をあらかじめ決めないで，プログラムの中で決まるようにしておいたら便利だろうと考える人がいるかもしれない．今時なプログラミング言語にはそういう機能が実装されている．Fortran でも，Fortran 90 以降には実装されている．

6.2 計算

6.2.1 代入

変数に値を代入するときは = を使う．= は左辺と右辺が等しいという意味ではないことに注意すること．

左辺=右辺で，右辺の値が左辺の変数の値となる．左辺は変数でなければならない．右辺は数値，あるいは数式．数式はその段階で数値計算可能でなければならない．

6.2.2 四則演算

四則演算は表 3 のように記述する．

表 3　Fortran での四則演算

演算	Fortran	数式	Fortran
加算	+	$x+y$	`x + y`
減算	-	$x-y$	`x - y`
掛け算	*	$x \times y$	`x * y`
割り算	/	$x \div y$	`x / y`
冪乗	**	x^y	`x**y`
括弧	( )	$\dfrac{2(x+y)}{3(a+b)}$	`(2.*(x+y))/(3.*(a+b))`

変数 x と変数 y の和を計算して，その値を変数 z に代入するなら，

コード 8　代入 1

```
1       z = x + y
```

以下のコードでは，1 行目が実行されると変数 x は 2 になり，2 行目が実行されると変数 y は 3 になる．3 行目が実行されると変数 x は 5 になる．4 行目が実行されると変数 y は 8 になる．

コード 9　代入 2

```
1       x = 2
2       y = 3
3       x = x + y
4       y = x + y
```

6.2.3　組み込み関数

三角関数などは標準で用意されているものを使うことができる (表 4)．関数の括弧の中に入る数のことを引数と呼ぶ．三角関数の引数の単位はラジアンである．度の単位を持った数値は，ラジアンに変換して三角関数に渡さなければならない．

表 4　Fortran の組み込み関数

Fortran	意味	Fortran	意味
`sqrt(x)`	$\sqrt{x}$	`exp(x)`	$\exp(x)$
`log(x)`	$\ln(x)$	`log10(x)`	$\log(x)$
`sin(x)`	$\sin(x)$	`asin(x)`	$\arcsin(x)$
`cos(x)`	$\cos(x)$	`acos(x)`	$\arccos(x)$
`tan(x)`	$\tan(x)$	`atan(x)`	$\arctan(x)$
`sinh(x)`	$\sinh(x)$	`cosh(x)`	$\cosh(x)$
`tanh(x)`	$\tanh(x)$	`abs(x)`	x の絶対値
`int(x)`	x の整数部	`float(n)`	整数 n を実数に変換
`max(x,y,z,...)`	$x, y, z, ...$ の最大値	`min(x,y,z,...)`	$x, y, z, ...$ の最小値

6.2.4　計算と変数の型

作法として，演算は変数の型を混ぜないことになっている[38]．例えば，整数と実数の足し算はしない．変数の型が異なるもの同士を使って演算をおこなう場合には，組み込み関数を用いて，変数の型を揃えてから演算をおこなう．整数を実数に変換するには，組み込み関数 `float` を使う．実数を整数に変換する組み込み関数 `int` を使ってもよい．

整数型と実数型を区別する Fortran において，数値に小数点 (`.`) が付いているかどうかは大きな違いとなる．小数点の付いた `1.` は実数であり，小数点の付かない `1` は整数となる．整数に小数点をつけてはならない．コード 10 では，`i = 1` と `y = 1.` は左辺と右辺の型が揃っていてお行儀のよい書き方になっているのに対し，`j = 1.` と `x = 1` は左辺と右辺の型が揃っておらずお行儀の悪い書き方になっている．コード 10 の 10 行目は，

[38] 変数の型が混ざった演算をするコードを実行すると，計算機は自動的に型を変換して演算を実行する．変数の型が混ざった演算をするコードを書いてしまっても，問題が生じることはそれほどないかもしれない (よく知らない)．

やってはいけない例である．9 行目と 10 行目の違いは，9 行目はそれより前に変数 a に値が代入されているのに対し，10 行目はそれより前に変数 c に値を代入されていない．= の右辺に出てくる変数には，それより前に値が代入されていなければならない．

コード 10　計算

```
1         integer i, j
2         real    x, y
3         real    a, b, c, d
4         i = 1
5         j = 1.
6         x = 1
7         y = 1.
8         a = 1.
9         b = a + 1.
10        d = c + 1.
```

コード 11 は，型に関連して生じるありがちなミスの例である．コード 11 を実行すると，変数 a，b，c はそれぞれ，4.5，1.0，3.0 になる．計算は前から順番に実行されるので，4 行目ではまず x**1 を計算した後に，/2 が計算される．すなわち，a には (x**1)/2 の結果である 4.5 が代入される．5 行目は，括弧の中の割り算が先に計算される．括弧の中の数はいずれも整数なので，計算された結果も整数となり，括弧の中の値は 0 になる．したがって，b には x**(0) の結果である 1.0 が代入される．6 行目でも括弧の中の割り算が先に計算される．括弧の中の数はいずれも実数なので，計算された結果も実数となり，変数 c には x**(0.5) の結果である 3.0 が代入される．

コード 11　計算

```
1         real    x
2         real    a, b, c
3         x = 9.0
4         a = x**1/2
5         b = x**(1/2)
6         c = x**(1.0/2.0)
```

6.2.5　単精度実数と倍精度実数

Fortran の実数には単精度と倍精度がある．実数と整数を混同してはいけないのと同じように，単精度実数と倍精度実数も混同してはならない．以下は単精度実数と倍精度実数の例である．

コード 12　単精度実数

```
1       real    PI, c, G
2       PI  = 3.14e0
3       c   = 2.99792458e8
4       G   = 6.67408e-11
```

コード 13　倍精度実数

```
1       real*8 PI, c, G
2       PI  = 3.14d0
3       c   = 2.99792458d8
4       G   = 6.67408d-11
```

上の例において, 変数 PI には 3.14 が, c には 2.99792458×10^{8} が, G には 6.67408×10^{-11} が代入される．倍精度実数の d0 を省略することはできないが，単精度実数は小数点を含んでいれば e0 を省略することができる．指数を使った方がわかりやすいので，指数を使うべきだと思うが，いちおう以下のように書いてもよいことになっている．

コード 14　単精度実数

```
1       real    PI, c, h
2       PI  = 3.14
3       c   = 299792458.
4       G   = 0.0000000000667408
```

6.2.6　function

自分で関数を定義することも可能.

6.3　条件分岐

6.3.1　do 文

do ループ，という呼び方をすることの方が多いかもしれない．do ループは「繰り返し」をするときに使う．例えば,

$$y = \sum_{i=1}^{10} a_i \tag{3}$$

の計算は，以下 (コード 15) のようにして実行することができる.

コード 15　繰り返し演算

```
1         y = 0.0
2         do 100 i = 1, 10
3           y = y + a(i)
4  100    continue
```

6.3.2 if 文

変数の値などの条件によって処理を変える場合に使う．Fortran では，次のようにして場合分けをおこなう (if の後に続く括弧の中の読み方は，6.3.3 節を見よ)．

コード 16　場合分け 1

```
1         if (i.eq.0) then
2             j = 1
3           else
4             j = 0
5         end if
```

この場合，i が 0 のとき j は 1 となり，それ以外で j は 0 になる．

もうひとつ例を出す．

コード 17　場合分け 2

```
1         if (x.ge.(0.0)) then
2             y = x
3           else
4             y = - x
5         end if
```

この 5 行は以下の 1 行と同じ結果を与える．

コード 18　場合分け 3

```
1         y = abs(x)
```

else の後に if を書くことで，さらに細かく条件分岐させることもできる．

コード 19　場合分け 4

```
1         if (v.lt.(33.0)) then
2             F = 0
3           else if (v.lt.(50.0)) then
```

```
4             F = 1
5           else if (v.lt.(70.0)) then
6             F = 2
7           else if (v.lt.(93.0)) then
8             F = 3
9           else if (v.lt.(117.0)) then
10            F = 4
11          else
12            F = 5
13        end if
```

これは，

$$F = \begin{cases} 0 & (v < 33.0) \\ 1 & (33.0 \leq v < 50.0) \\ 2 & (50.0 \leq v < 70.0) \\ 3 & (70.0 \leq v < 93.0) \\ 4 & (117.0 \leq v < 117.0) \\ 5 & (v > 117.0) \end{cases} \tag{4}$$

をコードで書いたものになっている[39]．

6.3.3 条件判断

FORTRAN 77 の世界では，大小符号や等号を以下のように書く．いちおう `eq` は **eq**ual，`ne` は **n**ot **e**qual，`lt` は **l**ess **t**han，`le` は **l**ess than or **e**qual であると思えば，覚えられなくもない (かもしれない)．

表 5　Fortran で扱う大小符号や等号

関係式	Fortran	関係式	Fortran
$a = b$	`a.eq.b`	$a \neq b$	`a.ne.b`
$a < b$	`a.lt.b`	$a \leq b$	`a.le.b`
$a > b$	`a.gt.b`	$a \geq b$	`a.ge.b`

[39] このコードは，風速 v (m/s) に対して藤田スケール F を与える．

6.4 入出力

6.4.1 write 文

write 文は出力をする．とりあえず，write(*,*) と書いて，後ろに出力したいものを書く．クォーテーション (') やダブルクォーテーション (") で括った部分は，そのまま出力される．括られていないものは変数と認識され，変数に格納されているデータが出力される．複数のものを出力するときは，, で区切って並べる．

コード 20　出力

```
1       write(*,*) t
2       write(*,*) t, p
3       write(*,*) t, 'temperature'
4       write(*,*) 'temperature', t
```

6.4.2 read 文

read 文を使ってデータを読み込むことができる．

コード 21　入力

```
1       read(*,*) x
2       read(*,*) i, j
```

read(*,*) に続けて変数を書くと，端末から入力されたものが変数 x に代入される．複数の値を読み込むときは，, で区切って変数を並べる．

6.4.3 open 文

open 文を使うと，ファイルから入力したりファイルに出力したりすることができる．

コード 22　入出力 1

```
1       open(unit=2,file='input.dat',status='old')
2       open(unit=3,file='output.dat',status='new')
3       do 100 i = 1, 10
4         read(2,*)  x
5         y = x + 1.0
6         write(3,*) y
7   100   continue
8       close(2)
```

```
9         close(3)
```

read 文の (*,*) を (2,*) にすると，入力は 2 番の論理装置からおこなわれる．1 行目の open 文は，2 番の論理装置を input.dat という名前の既存のファイルに設定している．同様に，write 文では (3,*) としたので，出力は 3 番の論理装置に対しておこなわれる．2 行目の open 文によって，3 番の論理装置は output.dat という名前の新規のファイルであると設定されている．

open 文で開いたファイルは，使用後に close 文を使って閉じる[40]．

6.4.4 読み込むデータの数を知らないとき

read 文にはもうひとつ，便利な機能がある．

コード 23 入出力 2

```
1         open(unit=2,file='input.dat',status='old')
2         open(unit=3,file='output.dat',status='new')
3         n = 2000
4         do 100 i = 1, n+1
5           read(2,*,end=900) x
6           y = x + 1.0
7           write(3,*) y
8   100   continue
9   900   continue
10        if (i.le.(n+1)) then
11            write(*,*) 'number of data =', i-1
12          else
13            write(*,*) 'number of data >' n
14        end if
15        close(2)
16        close(3)
```

このコードでは read 文の括弧の中に end=900 が追加されている．end= が指定されているとき，read 文がファイルの終りを読み込むと end= で指定した行番号にとぶ．上の例だと，do ループが n+1 回までまわり終える前に読み込みデータがなくなれば，行番号 900 番にとぶ．行番号 900 は do ループの外にあるので，この割り込み (行番号 900 にとぶこと) によって do ループは完結せずに終了となる．

[40] close 文で閉じるのをさぼっても問題は生じないかもしれないが，お行儀の問題として close 文で閉じることを推奨する．

行番号 900 の次の行 (10 行目) からはじまる 5 行は，データの数を出力する．データが n 個以下のとき，do ループは割り込みによって終了する．制御変数 i は，割り込み発生時の値を保持するので，i の値を見れば何回目のループがまわっているときにデータがなくなったのかわかる．i 回目のループでデータを読むことができなかったということは，データは i-1 個であったということである (11 行目はデータ数 i-1 を出力する)．データが n+1 個以上あるとき，do ループは割り込まれることなく最後までまわり，i は n+2 となって終了する．このときは，13 行目が実行される．

6.4.5 リダイレクション

リダイレクションは Fortran とは無関係であるのだが，入出力を安直におこなう方法として重宝するので，ここで説明しておく．

プログラムの実行時に > や < を使ってファイル名を指定することで，ファイルへの入出力をおこなうことができる．

```
$ ./a.out < input.dat > output.dat
```

この例では，プログラム中で read(*,*) が呼ばれるたびに，input.dat に書かれたデータが読み込まれる．また，プログラム中で write(*,*) が呼ばれるたびに，出力が output.dat に書き込まれる．ファイル名は input.dat や output.dat である必要はない．好きなファイル名を使ってよい．入力と出力は片方だけの指定でもよい．入力は端末からおこなって出力だけをファイルに書き込むのであれば，> output.dat だけを指定する．逆に，出力は端末におこなって入力をファイルからおこなうのであれば，< input.dat だけを指定する．

6.5 サブルーチン

サブルーチンはとっても便利である．

第 IV 部

gnuplot

7　図を描く

いろいろと計算した結果が出てくると，それを図に描いてみたくなる．gnuplot を使うと，手軽に図をつくることができる．

7.1　サンプルデータの準備

サンプルデータを用意する．まず作業用ディレクトリに移動する．

```
$ cd
$ cd work
```

サンプル・データをコピーして，展開する．

```
$ cp /home/george/student/gnuplot/plotdata.tar
$ tar xvf plotdata.tar
```

新しく plotdata というディレクトリが生成するので，中を見てみる．

```
$ cd plotdata
$ ls
```

いくつかのファイルが表示されるはず．試しにひとつ見てみる．

```
$ cat xy.dat
```

```
1   1
2   4
3   9
5  25
6  36
```

7.2 起動と終了

ターミナルウィンドウでコマンドを打って gnuplot を起動する．

```
$ gnuplot
```

gnuplot が起動し，オープニングのメッセージが表示されたあと，gnuplot のプロンプト (gnuplot>) が出る．

```
gnuplot>
```

gnuplot> のプロンプトに対して入力したものは，gnuplot によって処理される．gnuplot を終了するには，

```
gnuplot> exit
```

とする．gnuplot を終了したら，プロンプトは $ に戻る．

7.3 まず描いてみる

ファイルに書かれたデータを図に描いてみる．gnuplot が起動していない (プロンプトが gnuplot> になってない) なら，gnuplot を起動する．gnuplot で，ファイル xy.dat

に書かれたデータを描く．

```
gnuplot> plot 'xy.dat'
```

うまくいけば，新しいウィンドウが開いて図が描かれる．

7.3.1 スタイル

データを点で描く，折れ線描く，点と点の間を線で結ぶ，などなど，with でスタイルを指定する．

```
gnuplot> plot 'xy.dat' with points
gnuplot> plot 'xy.dat' with lines
gnuplot> plot 'xy.dat' with linespoints
```

他にもインパルス表示 (impulses)，誤差グラフ (errorbars)，棒グラフ (boxes)，などなど，いろいろあるので，描きたいものがあったらウェブで検索してみるとよい．

7.4 ヘルプ機能

この節は困ったことが生じたときに読む．困っていないなら，この節をとばして先に進む．

困ったときは Google などを使って検索することをお勧めする．gnuplot を使っている人はそこそこの数いるので，案外簡単に解決方法が見つかったりする．

ウェブで検索する方が簡単であると思うが，いちおう gnuplot のヘルプ機能についても説明しておく．gnuplot のヘルプ機能は以下のように起動する．

```
gnuplot> help
```

最初に概要説明が表示される．複数ページにわたるときは，最終行が

```
:
```

となる．『Enter』キーを押すと，1 行ずつ進む．『スペース』キーを押すと，1 ページずつ進む

```
(END)
```

となったら，『Q』のキーを押して先に進む．たくさんの項目が表示され，以下のようなプロンプトが表示される．

```
Help topic:
```

表示された中から，知りたい項目を選んで入力する．例えば，terminal について調べるなら

```
Help topic: terminal
```

また，いくつかの項目が表示され，プロンプトは以下のようになる．

```
Subtopic of terminal:
```

これは，terminal の下のサブトピックを選べ，ということである．このように，項目は階層構造で整理されているので，見たいものが見つかるまで選択をくり返して，階層を降りていく．階層を上がるときは，何も入力せずに『Enter』キーを打つ．最上位の階層で『Enter』キーを打つと，プロンプトが gnuplot に戻る．

7.5 データファイル

ひとつのデータポイントを表わすデータは 1 行の中に書く．空白，またはタブで区切る

データの区切りとして，空白またはタブ，以外のものを使うときは，

```
gnuplot> set datafile separator ’,’
```

のようにして，区切り文字を指定する．上の例は区切り文字として「,」を指定している．

空行には意味がある空行が何をするのか知りたい人は，調べてみること．初心者は，データファイルに空行を入れないようにするのがよい．

改行コードのせいで，gnuplot がデータの読み込みに失敗することがよくある．なんだかわからないけどうまくいかない，というときは，データファイルの改行コードを変換してみると，うまくいくことがある．改行コードの変換は，

```
$ nkf -Lu --overwrite xy.dat
```

nkf は文字コードの変換をするコマンド．この例では，ファイル xy.dat の文字コードを変換する．オプションの -Lu は，改行コードを Unix にする．オプションの --overwrite は結果を上書きする．元ファイル (xy.dat) を残しておく必要がある場合は，--overwrite を使ってはならない．

7.6 図の体裁を整える

図の体裁を整えるときは，set コマンドを使って設定を指定する．gnuplot の set コマンドは，これから描かれる図に対して効力を発揮する．描いた図には影響を及ぼさない．例えば，

```
gnuplot> plot ’xy.dat’
```

として図を描いてみて，横軸の範囲を変更したくなったとする．横軸の範囲を指定するため，

```
gnuplot> set xrange [2:4]
```

と入力する．横軸の範囲を指定するのはこれで正しいが，図の横軸の範囲は変わらない．横軸の範囲を変更した図を描きたいなら，再度 plot を実行する必要がある．

```
gnuplot> plot 'xy.dat'
```

新しく描かれた図の横軸の範囲は set xrange で指定したものになる．

再描画は，

```
gnuplot> replot
```

としてもよい．replot は直前に実行した plot コマンドを再実行するコマンドである (ファイル名の指定などを省略することができる (ただそれだけの) コマンドである)．

7.6.1 軸の範囲を指定する

横軸と縦軸の範囲を指定するときは

```
gnuplot> set xrange [0:5]
gnuplot> set xrange [1:100]
```

とする．

軸を反転させるときは

```
gnuplot> set xrange [5:0]
gnuplot> set yrange [100:1]
```

設定を解除するときは

```
gnuplot> unset xrange
gnuplot> unset yrange
```

7.6.2 軸の目盛りを変更する

3 つの数字で，開始位置，目盛りの間隔，終了位置，を設定する．

```
gnuplot> set xtics 3, 0.2, 4
```

小目盛りを表示するときは

```
gnuplot> set mxtics 5
```

数字は大目盛りの間を分割する数である．5 を指定すると 5 分割することになるので，小目盛りが 4 つ表示される．

7.6.3 目盛りの書式

目盛りを 2×10^3 のように表示するなら，

```
gnuplot> set format x "%t{/Symbol \264}10^{%T}"
```

ここで，%t は基数が 10 の仮数，%T は基数が 10 の指数，である．{/Symbol \264} は「×」である．

小数点以下何桁まで書くのかは，% と t の間に $.n$ と書いて指定する．

```
gnuplot> set format x "%.2t{/Symbol \264}10^{%T}"
```

これは $n = 2$ を指定しているので，2000 は 2.00×10^3 になる．

小数点以下を書かないなら，

```
gnuplot> set format x "%.0t{/Symbol \264}10^{%T}"
```

7.6.4 タイトル

図にタイトルをつける．

```
gnuplot> set title "title"
```

7.6.5 軸の説明

x 軸と y 軸に説明をつける．

```
gnuplot> set xlabel "xlabel"
gnuplot> set ylabel "ylabel"
```

7.6.6 グリッドの表示

グリッドを表示する (表示しない)．

```
gnuplot> set grid
gnuplot> unset grid
```

7.6.7　対数グラフ

対数を使う (使わない).

```
gnuplot> set logscale x
gnuplot> set logscale y
gnuplot> unset logscale x
gnuplot> unset logscale y
```

7.6.8　列を選ぶ

gnuplot は 1 行でひとつのデータポイントを表わす．1 行の中には 3 つ以上の列があってもよい．1 列目を横軸，3 列目を縦軸にして，図を描くときは，

```
gnuplot> plot 'xy.dat' using 1:3
```

using の指定を省略したときは，1:2 が指定されたことになる．縦軸と横軸を入れ替えた図を描くときは

```
gnuplot> plot 'xy.dat' using 2:1
```

とする．

using で 0 列目を指定すると，行を指定したことになる．すなわち，

```
gnuplot> plot 'xy.dat' using 0:3
```

は，3 列目にある値を順番に，横軸 0, 1, 2, ... の位置にプロットする．

7.6.9　データの演算

データを 2 倍にしたり，データから 5 を引いたものを描きたくなったら，

```
gnuplot> plot 'xy.dat' using 1:($2*2.0)
gnuplot> plot 'xy.dat' using 1:($2-5.0)
```

変数を使うこともできる

```
gnuplot> a=2.0
gnuplot> plot 'xy.dat' using 1:($2*a)
```

データ同士の演算もできる

```
gnuplot> plot 'xy.dat' using 1:($2+$3)
```

7.6.10　行をとばす

every を使うとデータの特定の行を選んでプロットすることができる．

```
gnuplot> plot 'xy.dat' every 3
gnuplot> plot 'xy.dat' every ::5
```

最初の例は，3 つ毎に (1, 4, 7, ... の行を) プロットする．2 つ目の例は，最初の 5 行をスキップする場合．

every を使うと，他にもいろいろなことができる．興味のある人は調べてみるとよい．

7.6.11 色の指定

色を指定する．

```
gnuplot> plot 'xy.dat' lc rgb "red"
```

lc rgb に続けて色を書く．上の例だと，赤色 (red) でプロットされる．gnuplot で使うことのできる色は，ウェブで検索して調べる．よくある色はだいたい使うことができる．

7.6.12 時間と日付

時間や日付を使うときは，set xdata time として，日時の書式を set timefmt で指定する．そして，plot は必ず using で列を指定する．

```
gnuplot> set xdata time
gnuplot> set timefmt "%Y/%m/%d\t%H:%M"
gnuplot> plot 'data.dat' using 1:5
```

表 6　日時を表わす書式

%d	何日, 1-31	%H	何時, 0-24
%m	何月, 1-12	%M	何分, 0-60
%y	何年, 0-99	%S	何秒, 0-60
%Y	何年, 4 桁	%b	月名 (英語) の 3 文字省略形
%j	1 年の何日目, 1-365	%B	月名 (英語)
\t	タブ	スペース	0 個あるいは 1 つ以上の空白文字列

xdata time を指定したときの，xrange や xtics の設定．

```
gnuplot> set xdata time
gnuplot> set timefmt "%Y/%m/%d %H:%M"
gnuplot> set format x "%H:%M"
gnuplot> set xtics "00:00", 7200, "24:00"
gnuplot> plot 'data.dat' using 1:5
```

xtics の目盛りの間隔は秒の単位で指定する．上の例は 7200 秒を指定しているので，2 時間おきに目盛りが描かれる．

7.6.13　3 次元表示

3 次元で図を描く．

7.7　図の保存

作成したグラフを保存するには

```
gnuplot> set terminal pngcairo
gnuplot> set output 'xy.png'
gnuplot> replot
```

set terminal は，出力先を指定する．この例では PNG 形式の画像を出力するようにしている．

set output で出力先のファイル名を指定する．この例では，xy.png という名前のファイルを指定している．

replot は，再描画の命令である．ついさっき描いたのと同じものを再度描画する．

7.8　スクリプトファイル

一連のコマンドをテキストファイルに書き込んでおいて，それを読み込むことで一連のコマンドを実行することができる．一連のコマンドを書き込んだファイルは，スクリプトファイルと呼ばれる．gnuplot でスクリプトファイルを使うときは，load に続いてスク

リプトファイルのファイル名を書く．

```
gnuplot> load 'hoge.gp'
```

この例だと，hoge.gp という名前のファイルが読み込まれて，その中に書かれているコマンドが実行される．gnuplot のスクリプトファイルの拡張子は，gp でなくてもよい．

以下は，gnuplot で描いた図を gnuplot.png という名前のファイルに保存するスクリプトである．

コード 24　savepng.gp

```
1 set terminal pngcairo
2 set output 'gnuplot.png'
3 replot
```

gnuplot で図を作った後，

```
gnuplot> load 'savepng.gp'
```

とすると，スクリプトを実行する直前に描いた図が gnuplot.png に保存される．

スクリプトファイルに書いた命令は上から順番に実行される．先頭が「#」になっている行は，コメントとして無視される．

スクリプトファイルの使い方はいろいろである．例えば，いつも使う設定をスクリプトファイルに書いておいて，最初にスクリプトファイルを読み込む．そうすると，毎回やる作業を省略して，いつも同じ環境で作業を開始することができる．

図を作成するときに使ったコマンドの全てをスクリプトファイルに書いておく，というのもよい．後でデータがちょこっとだけ書き変わったとき，また最初から全てのコマンドを打つのは面倒である．スクリプトファイルに書いておけば，ファイルを読み込むだけで同じ図をすぐに作ることができる．

7.9　シェルスクリプトで図をつくる

図を生成するコマンドを全てファイルに書き込んでおけば，gnuplot を起動せず，シェルから直接に図のファイルを生成することもできる．

コード 25　sin.sh

```
1 #!/bin/bash
2 gnuplot <<EOF
3 set terminal pngcairo
4 set output 'sin.png'
5 set xrange [-pi:pi]
6 set yrange [-1:1]
7 plot sin(x)
8 EOF
```

コード 25 はシェルで実行するコマンドを書いたテキストファイル (シェルスクリプト) である．シェルスクリプトの実行は，

```
$ ./test.sh
```

正しく実行されれば sin.png という名前のファイルが生成する．

1 行目の #!/bin/bash は，シェルスクリプトのおまじないである．2 行目の gnuplot <<EOF で gnuplot が呼び出されて，次に EOF が出てくる (8 行目) までの範囲が gnuplot によって実行される．EOF 行までの間に空行を入れてはいけない．

using を使って演算をおこなうときは，シェルスクリプトの変数と混同されないようにするため，\ (バックスラッシュ) をつける．

コード 26　xy12.sh

```
1 #!/bin/bash
2 gnuplot <<EOF
3 set terminal pngcairo
4 set output 'xy12.png'
5 plot 'xy.dat' using 1:(\$1+\$2)
6 EOF
```

7.10　データのフィッティング

data.dat に書かれたデータを任意の関数でフィッティングする．

まず関数を定義する．例えば 2 次関数でフィッティングするなら，

```
gnuplot> f(x)=a*x*x+b*x+c
```

ここでは関数を f(x) としたが，関数の名前は違うものでもよい．

次にフィッティング．フィッティングの前に，関数に含まれる係数の初期値を与えてから (初期値がダサいと収束が遅くなったり，収束しなかったりすることもある)，フィッティングする．

```
gnuplot> a=1
gnuplot> b=2
gnuplot> c=10
gnuplot> fit f(x) 'data.dat' using 1:2 via a,b,c
```

最初の 3 行は初期値を与えている．最後の fit でフィッティングの計算が実行されて，途中経過も含めて結果が表示される．フィッティングの結果は fit.log というファイルにも保存される．結果をグラフ化するには

```
gnuplot> plot f(x), 'data.dat' using 1:2
```

特定の範囲にあるデータだけを使ってフィッティングするときは，fit に続いて範囲 [] で指定する．

```
gnuplot> fit f(x) [0:1][] 'data.dat' using 1:2 via a,b,c
```

この例だと，x (uxing 1:2 としているので，x は data.dat の 1 列目) が [0:1] の範囲にあるデータのみを使ってフィッティングを行う．2 つ目の [] の中身を空欄にしているので，y (uxing 1:2 としているので，y は data.dat の 2 列目) の値によってデータの選別はおこなわない．

7.11　お手軽アニメーション

お手軽なアニメーションの作り方.

第 V 部
シェルスクリプト

8　シェル

8.1　コマンドラインシェル

シェルスクリプトの前に，まず「シェル (shell)」である．

Unix では，シェルと呼ばれる種類のソフトウェアを使って，計算機に命令を伝える．シェルは大別すると，コマンドライン・シェル (command-line shell) と，グラフィカル・シェル (graphical shell) に分けられる．前者は主にキーボードからの文字入力で操作をおこなう CLI (command line interface)[41]の形態をとったもの，後者は画面上に表示されたアイコンなどをマウスや画面へのタッチによって操作する GUI (Graphical User Interface) の形態をとったもの，である．

ここでは，CLI であるコマンドライン・シェルを用いる．CLI と GUI，だいたいのことはどちらを使ってもできるのだが，ある種の作業は GUI を使っておこなうと煩雑になり過ぎてしまい，CLI を使わないとやってられない (ということがある)．なので，CLI を使う．CLI を使うためには，コマンドとその構文などを知っている必要があるため，最初は少し覚えることが多く，そのことが CLI をとっつきにくくしていると思われる[42]．とはいえ，頻繁に使うものはそれほど多くないので，しばらく使っていれば必要なことは自然に覚えてしまうだろう．

ひとまず CLI とお付き合いしてみる．そのうちに，とっつきにくさは消えて，CLI の良さが分かるようになる (と思う)．

[41] CLI は，キーボードなどから入力される文字列の行 (ライン) を解釈して，OS やプログラミング言語処理系などに渡す，インターフェースである．

[42] GUI はなんとなく使い方がわかって，なんとなく使えてしまう．CLI は知らないと手も足も出ない．

OS(オペレーティング・システム)

ソフトウェアは大きく分けて，基本ソフトウェアと応用ソフトウェアの 2 つに分けられる．基本ソフトウェアは OS (Operating System) と呼ばれるもので，Linux，Windows，Mac OS などが相当する．OS は計算機を利用する上での基本的な機能をまとめたソフトウェアで，ハードウェアの管理・制御をおこなう．例えば，キーボードを使った入力や，ディスプレイに文字や絵を表示するといった出力は，OS によって管理・制御されている．

Chrome，Firefox，Thunderbird，MS Word，PowerPoint，Acrobat Reader，などのアプリケーションは，応用ソフトウェアである．アプリケーション・ソフトウェアは，入力や出力といった基本的な処理をおこなう時，OS を呼び出して代わりにやってもらっている．すなわち，アプリケーションは，ハードウェアとのやりとりを OS に任せることで，楽をしている．もし，アプリケーションがハードウェアを直接制御するならば，アプリケーションはハードウェアに合わせて作られなければならなくなるが，ハードウェアとのやりとりを OS に任せることにすれば，アプリケーションはハードウェアが何であるかを気にすることなく，OS とのみ整合するように作ればよいことになる．分業することで，アプリケーションの開発がやりやすくなっているのである．OS と整合するように作られたアプリケーションは，別の OS で動かすことができない．アプリケーションのダウンロードするときに OS を選ぶのは，このことがあるからである．

カーネル (核) とシェル (殻)

Unix 系 OS の中核はカーネル (kernel) と呼ばれ，カーネルがハードウェアの管理・制御，プロセス管理などをおこなっている．ユーザはカーネルに直接命令することはできない．ユーザが計算機に何かやらせるときには，シェル (shell) を使う．シェルは，ユーザとカーネルのインターフェースとなるソフトウェアで，ユーザの命令を聞いてカーネルに伝える．シェルという名前の由来は，ユーザとカーネルの間にあって核 (カーネル) を包む殻のように動作しているから．

8.1.1 シェル芸

巷ではシェル芸なるものが流行っているらしい．上田隆一先生による「シェル芸の定義バージョン 1.1」(`https://b.ueda.tech/?page=01434`) は，

マウスも使わず，ソースコードも残さず，GUI ツールを立ち上げる間もなく，あらゆる調査・計算・テキスト処理を CLI 端末へのコマンド入力一撃で終わらすこと．あるいはそのときのコマンド入力のこと．

1 行コードを書いて処理を終えることを「ワンライナー (one-liner program)」と言ったりする[43]．なんでも一撃で終わらせるシェル芸というのは，ワンライナーを極めること，なのだと思う (たぶん)．

一般の人はシェル芸などしなくてもよいと思うが[44]，シェル芸で使われる技術のいくつかは普通の人がシェルを使う上でも役に立つ．興味があれば，シェル芸について調べてみるのもいいかもしれない．

8.2 パイプ

シェルは，| を使うことで，コマンドの実行結果を次のコマンドの入力にすることができる．

```
$ ls /bin | less
```

この例では，`ls /bin` の出力を，`less` に渡している．ファイルがたくさんある時，`ls` だと結果の一部がウインドウの外に出てしまう．`ls` の結果を `less` に渡してやることで，すべてのファイルを見ることができる．

パイプは，コマンドの出力を加工するときにも使われる．例えば，行の数を数えるコマンド `wc -l` をパイプでつないだ以下の例は，拡張子が `jpg` であるファイルの数を教えてくれる．

```
$ ls *.jpg | wc -l
```

パイプは何段でもつなぐことができる．前のコマンドの出力をパイプで受け取り，それ

[43] one-liner と言うと，one-line joke (短いジョーク，気の利いた言い回し) を指すような気がする．シェル芸もジョークの一種なのかもしれない．

[44] シェル芸を使いこなす人を見て，「恰好いい」と思う人と「キモい」と感じる人がいるらしい．はしもとは前者であるが，後者の方が圧倒的な多数派であるような気がする．

を加工して，またパイプで次のコマンドに受け渡す，といったことができる[45].

8.3 リダイレクト

シェルは,

- コマンドの出力を任意のファイルに書き出す
- コマンドへの入力を任意のファイルから行う

ことが可能である．これをリダイレクトと呼ぶ.

```
$ ls *.txt > filelist
$ cat < filelist
```

1 行目は，.txt で終わるファイルのリストをつくって filelist という名前のファイルに格納する．2 行目は，filelist を入力として，そこに書かれたすべてのファイルを標準出力に出力する.

入力と出力は同時使用可能.

```
$ cat < filelist.txt > text.txt
```

ファイルの末尾に追加するときは，>> を使う.

```
$ echo "hoge" > out
$ cat out
hoge
$ echo "hero" > out
$ cat out
hero
```

[45] シェル芸はパイプを多用する．ちょっと複雑なことをしたいと思ったら，ひとつのコマンドで結果が得られるということはまずなくて，いくつかのコマンドを組み合わせる必要があるからである.

```
$ echo "hoge" > out
$ echo "hero" >> out
$ cat out
hoge
hero
```

出力先として指定された名前のファイルが存在するとき，> はファイルを上書きする (古いファイルを消去して新しいファイルを作る) が，>> はファイルの末尾に追加する．

9　シェルスクリプト

9.1　スクリプト

CLI は，あらかじめコマンドを書き込んだファイルを作成しておいて，そのファイルを CLI に読ませることによってコマンドを実行することができる．このコマンドを書き込んだファイルはスクリプト・ファイル (script files) と呼ばれる[46]．型どおりの決まった作業は，スクリプト・ファイルを作ることでいちいち入力する手間を省くことができる[47]．

スクリプトを書くのは簡単で，対話モードにおいて入力するのと同じものを，ただファイルに書くだけである．スクリプトは，1 行だけでなく，複数の行を書くことができる．スクリプトを実行すると，ファイルに書いた一連のコマンドが上から順番に実行されていく (これを順次構造という)．スクリプトは，反復構造，選択構造，変数，などを使って書くことができる．これらの制御構造と変数を使うことで，スクリプトに汎用性を持たせることができる．

構造化定理 (structure theorem)

任意の流れ図 (フローチャート) は，3 つの基本構造 (順次・反復・選択) だけの組み合わせによって記述することができる．

- 順次　上から下へ順に逐次処理をおこなっていく．
- 反復　条件が満たされている間，処理を繰り返す．
- 選択　条件が成立するか否かで処理を分ける．

スパゲッティ・プログラム (spaghetti program)

流れ図がごちゃごちゃしていて全体を把握することが難しいプログラムを，スパゲッティ・プログラムと呼ぶ．スパゲッティ・プログラムは，良くないという意味で使われる．ソースが絡まったスパゲッティは美味しいが，絡まったプログラムは良くないのである．

[46] スクリプト (script) は「台本」である．台本を渡された計算機は，台本通りに作業を遂行する．

[47] 計算機にアドリブはない．台本で決められた通りのことしかしない (決められた通りのことしかしないのが計算機の良いところである)．

コードを書く理由 (1.1 節) と重複する部分もあるが，スクリプトを書く理由について述べてみる．スクリプトを書くことの利点はいくつかあると思うが，大きな利点は次の 2 つであると思う．ひとつは，同じことをしたくなったときに，もう一度入力する必要がないということ．スクリプトを実行すれば，前回やったのと同じことがたちまち実行される．1 回ちゃんと書いたら，もう 2 度と入力しないでよい．どんなに長いコマンドであっても，完全に同じことを再実行することができる．もうひとつは，何をやったのか記録が残ること．あとで見直しをしなければならなくなったとき，記録が残っていることは非常に役に立つ．不幸にしてスクリプトに間違いが見つかったとしても，修正してすぐ再実行することができる．この 2 つだけでも，スクリプトを書く (CLI を使う) 理由として十分なものであると思う．

と書いたものの，初心者の多くはスクリプトを書く必要など感じないかもしれない．「あらかじめできるようになっておくと，いざという時の役に立ちます」ということを言う人はよくいるが，いざという時になってから勉強すればよいような気がする．そもそも，いざという時は来ないかもしれない．スクリプトを書かねばならないという状況にない人は，いつか役に立つなどといった功利的な理由ではなく，「スクリプトを書くのは楽しい」というのを，手を動かす原動力にできたらよいと思う．なんであっても，できる，というのは楽しいものであると思うから[48]．

9.2 シェルスクリプトを使ってみる

コード 27(makebackup.sh) は，hoge.txt というファイルのバックアップを生成するスクリプトである．バックアップとして作られるファイルには，ファイル名にバックアップを作成した日時が入る．

コード 27 makebackup.sh

```
1 #!/bin/bash
2 YMDHM=‘date ’+%Y%m%d%H%M’‘
3 cp hoge.txt hoge_${YMDHM}.txt
```

スクリプトを動かして，動作確認してみる．まず，hoge.txt を作成し，次いでスクリプトを動かしてそのバックアップを作成する．

[48] できないからつまらない，というのもよくあることである．

```
$ echo tako > hoge.txt
$ ls
hoge.txt    makebackup.sh
$ cat hoge.txt
tako
$ ./makebackup.sh
$ ls
hoge.txt    hoge_201809120942.txt    makebackup.sh
$ cat hoge_201809120942.txt
tako
```

最初の 3 つのコマンドは準備のためのもの．echo コマンドとリダイレクトを使って，hoge.txt というファイルに tako と書き込む．ls でファイルが生成されたことを確認．cat でファイルの中身を確認．

4 つめのコマンドで，バックアップのスクリプトを実行．

その後は動作確認．ls でバックアップが生成したことを確認．この例では，ファイル名に _201809120942 が付加されたファイルが生成している[49]．違う日時にスクリプトを実行すれば，違う名前のファイルが生成する．最後に cat でバックアップの中身を確認している．

9.3 makebackup.sh を解読する

スクリプトは基本的に前から順番に実行される．

1 行目は，このスクリプトは /bin/bash で実行してください，ということを言っている．シェルには，sh，bash，csh，tcsh，zsh，などなど，いろいろなものがあるが，ここでは bash を使う．シェルによって規則が違っていたりするので，2 行目以降は 1 行目で指定したシェルの規則に従って書かれなければならない．

2 行目は，YMDHM という名前の変数を設定し，それに年月日時分を代入している．ここ

[49] どうでもいいけど，9 月 12 日と 9:42 は，デジタル表示の時にアップルが好んで使う日時である (アナログの Apple Watch では，10 時 9 分 30 秒が使われる)．時計メーカーはそれぞれ贔屓にしている時刻があるらしく，アナログ時計だと，セイコーは 10 時 8 分 42 秒，シチズンは 10 時 9 分 35 秒，カシオは 10 時 8 分 37 秒，を使うらしい．

ではコマンド置換という機能が使用されている．コマンド置換とは，バッククォート ‘ で囲まれた部分をコマンドと解釈してコマンドの出力[50]で置き換える，というものである．date は現在の時刻を取得するときに使われるコマンドで，+%Y%m%d%H%M はコマンド出力の形式を指定している[51]．

3 行目は，ファイルのバックアップ (コピー) を作っている．バックアップのファイル名には，バックアップ生成時の年月日時分が埋め込まれている．

この一連の作業はもちろん手作業でおこなうことも可能である．スクリプトの中身をそのまま入力したらよい．

```
$ YMDHM=‘date ’+%Y%m%d%H%M’‘
$ cp hoge.txt hoge_${YMDHM}.txt
```

手で入力するときは，変数を経由しない人が多いかもしれない．

```
$ date ’+%Y%m%d%H%M’
201809120942
$ cp hoge.txt hoge_201809120942.txt
```

いずれであっても，手で入力していることとスクリプトに書いてあることは (ほとんど) 変わらない．シェルスクリプトを書くことは，手で入力することをあらかじめファイルに書いておくことである，ということがわかるだろう．

9.4 アクセス権

スクリプトを実行するためには，ファイルのアクセス権 (パーミッション) がしかるべく設定されている必要がある．ファイルのアクセス権は，ls を -l のオプション付きで実行して確認することができる．

[50] 末尾の改行は削除される．

[51] なんとなくわかるかもしれないが，%Y は西暦 (4 桁)，%m は月，%d は日，%H は時，%M は分，である．

```
$ ls -l
-rwxr--r--  1 george  staff      83 Sep 12 09:41 makebackup.sh*
```

スクリプトが実行可能であるためには，左から 4 つ目の文字が x になっている必要がある．左から 4 つ目の文字が - になっているときは，以下の手順でアクセス権を変更すれば，スクリプトを実行可能にすることができる．

```
$ chmod u+x makebackup.sh
```

特別なことをしていなければ，新規で作成したファイルのアクセス権の「実行」は不許可になっている．エディタでスクリプトを作ったら，動かす前に手動でアクセス権を変更する必要がある．

9.5 シェルの引数として実行する

スクリプトはシェルの引数として実行することもできる．このとき，スクリプトファイルのアクセス権として「実行」が許可されている必要はない (「読み出し」は許可されている必要がある)．

```
$ bash makebackup.txt
```

シェルの引数として実行するときは，スクリプトの先頭におまじない (#!/bin/bash) を書かなくてもよい．

コード 28　makebackup.txt

```
1 YMDHM=‘date ’+%Y%m%d%H%M’‘
2 cp hoge.txt hoge_${YMDHM}.txt
```

10　シェルの基本的な技

シェルの基本的な技について説明する．シェルスクリプトは，これらの組み合わせで作られる．

10.1　変数

変数に値を代入するときは，変数と値を = でつなぐ．= の右にあるものが，左に代入される．= の前後に空白を入れてはいけない．空白を含む文字列を代入するときは，クォーテーションで括る．

変数の値を参照するときは，変数の先頭に $ をつける．代入するときは $ をつけない．

コード 29　var.sh

```
1 #!/bin/bash
2 MONTH=Sep
3 DAY=12
4 echo $MONTH
5 echo $DAY
6 MD1='${MONTH} ${DAY}'
7 MD2="${MONTH} ${DAY}"
8 echo $MD1
9 echo $MD2
```

これを実行すると，

```
$ ./var.sh
Sep
12
${MONTH} ${DAY}
Sep 12
```

ダブルクォーテーション (") で囲んだ文字列では，変数が展開される．展開についての説明は 10.6 節．

10.1.1 配列

配列の使い方.

10.2 算術演算

10.2.1 四則演算

整数の四則演算は，算術式を `$((` と `))` で囲む．四則演算は `expr` も使う方法もあるが，計算に時間がかかるので，二重括弧を使う方がよさそう[52].

二重括弧の中は，変数に `$` をつけなくてもよい．すなわち，`$(( x + y ))` と書いたら，変数 `x` と変数 `y` の和が計算される．もちろん `$(( $x + $y ))` と書いてもよい．

二重括弧を使った四則演算では，小数は使えない (エラーになる)．また，除算の戻り値は整数 (小数点以下切り捨て) になる．

表 7　シェルの四則演算

bash	意味
`$(( x + y ))`	$x + y$ (和)
`$(( x - y ))`	$x - y$ (差)
`$(( x * y ))`	$x \times y$ (積)
`$(( x / y ))`	x を y で割った商
`$(( x % y ))`	x を y で割った余り
`$(( x ** y ))`	x^y (べき乗)

10.2.2 算術比較

数値を比較して，真か偽かを返す.

10.3 入力

10.3.1 引数

スクリプトを実行するときに値を渡すことができる．スクリプトに渡す値は引数 (ひきすう) と呼ばれる．引数を参照するときは，`$1`，`$2`，`$3`，... のように参照する．また，`$@`

[52] 二重括弧が使えない環境もあるらしいので，`expr` を使う方が汎用性は高いらしい.

表 8　シェルの算術比較

bash	意味
`$a -eq $b`	$a = b$ の場合に真
`$a -ne $b`	$a \neq b$ の場合に真
`$a -gt $b`	$a > b$ の場合に真
`$a -lt $b`	$a < b$ の場合に真
`$a -ge $b`	$a \geq b$ の場合に真
`$a -le $b`	$a \leq b$ の場合に真

ですべての引数を，`$#` で引数の数を，それぞれ参照することができる．

コード 30　arg.sh

```
1 #!/bin/bash
2 echo $#
3 echo $1
4 echo $2
5 echo $3
6 echo $4
7 echo $@
```

実行してみる

```
$ ./arg.sh apple banana carrot
3
apple
banana
carrot

apple banana carrot
```

最初は引数の数 3 を表示．次は 1 つ目の引数 `apple`，その次は 2 つ目の引数 `banana`，その次は 3 つ目の引数 `carrot` を表示．4 つめの引数はないので空行になる．最後は全ての

引数を表示.

もうちょっと意味のあるスクリプトを考えてみる．9.2 節に出てきたスクリプト (コード 27) は，hoge.txt のバックアップしかできない．引数を使うことで，任意のファイルのバックアップを作成することができるようにする.

コード 31　makebackup2.sh

```
1 #!/bin/bash
2 NAME1=${1%.*}
3 NAME2=${1##*.}
4 YMDHM=‘date ’+%Y%m%d%H%M’‘
5 cp $1 ${NAME1}_${YMDHM}.${NAME2}
```

実行.

```
$ ls
hero.txt    makebackup2.sh
$ ./makebackup2.sh hero.txt
$ ls
hero.txt    hero_201809120942.txt    makebackup.txt
```

10.3.2　read

read は入力されたものを読み込んで変数に格納する.

コード 32　read.sh

```
1 #!/bin/bash
2 echo ’Enter your name’
3 read name
4 echo "Hello $name !"
```

10.4 選択構造

10.4.1 if

条件によって処理を変える．シェルスクリプトは，if で始まって fi で終わる．
[の後ろと] の前には空白を入れる．

コード 33 if1.sh

```
1 #!/bin/bash
2 if [ $1 = $2 ]; then
3     echo same
4   else
5     echo different
6 fi
```

分けたものをさらに分けるときは elif を使う．

コード 34 if2.sh

```
1 #!/bin/bash
2 if [ $1 -gt $2 ]; then
3     echo $1 '>' $2
4   elif [ $1 -eq $2 ]; then
5     echo $1 '=' $2
6   else
7     echo $1 '<' $2
8 fi
```

真偽を判定するコマンド

シェルでは，0 が正，0 以外が偽，である．
真偽を判定するコマンドに test がある．test は，与えられた式が真のとき 0 を返し，偽のとき 1 を返す．
if などに使われる [も真偽を判定するコマンドである[a]．test と同じ働きをする．

[a] コマンドっぽく見えないところが恰好いい．

10.4.2 case

case で場合分けすることもできる．case は esac で閉じる．

コード 35 case1.sh

```
#!/bin/bash
case $# in
  0) echo 'I don not have a favorite thing.'        ;;
  1) echo 'My favorite thing is' ${1}.              ;;
  2) echo 'My favorite things are' ${1} and ${2}.   ;;
  *) echo 'I have a lot of favorite things.'        ;;
esac
```

$# は引数の数なので，引数の数に応じて出力が変わる．* はワイルドカードで，すべての文字列にマッチする．

ちなみに，このスクリプトは case の中を逆順に並べると機能しない．

コード 36 case2.sh

```
#!/bin/bash
case $# in
  *) echo 'I have a lot of favorite things.'        ;;
  2) echo 'My favorite things are' ${1} and ${2}.   ;;
  1) echo 'My favorite thing is' ${1}.              ;;
  0) echo 'I don not have a favorite thing.'        ;;
esac
```

何を入力しても出力は同じになる．理由はわかりますよね？

10.5 反復構造

10.5.1 while

while は条件が満足されている間，do と done の間を繰り返す．

コード 37 while.sh

```
#!/bin/bash
```

```
2 i=1
3 while [ $i -le 10 ]; do
4   echo $i
5   i=$(( i + 1 ))
6 done
```

10.5.2 until

`until` は `while` の逆．条件が満足されるまで，`do` と `done` の間を繰り返す．

コード 38　until.sh

```
1 #!/bin/bash
2 i=1
3 until [ $i -gt 10 ]; do
4   echo $i
5   i=$(( i + 1 ))
6 done
```

10.5.3 for

与えられた値を代わりばんこに代入して，`do` と `done` の間を繰り返す．

コード 39　for.sh

```
1 #!/bin/bash
2 for words in 'apples bananas carrots'; do
3   echo 'I like' ${words}.
4 done
```

10.6 展開

展開は以下の順序でおこなわれる．

1. ブレース展開 (brace expansion)
2. チルダ展開 (tilde expansion)
3. パラメータ・変数展開 (parameter and variable expansion)
4. 算術式展開 (arithmetic expansion)

5. コマンド置換 (command substitution)
6. 単語分割 (word splitting)
7. パス名展開 (pathname expansion)
8. クォートの削除 (quote removal)

これらの展開がおこなわれた後，コマンドに引数が分割された形で渡されて実行される．

10.6.1 ブレース展開

シーケンス展開

```
$ echo {a..c}
a b c
$ echo {0..3}
0 1 2 3
$ echo {0..3}a
0a 1a 2a 3a
```

カンマ区切り展開

```
$ echo {apple,orange}
apple orange
```

10.6.2 チルダ展開

チルダ文字 ~ はシェルパラメータ HOME に置き換えられる．HOME が設定されていない場合は，シェルを実行しているユーザのホームディレクトリに置き換えられる．

```
$ echo ~
/home/george
$ echo ~/work
```

```
/home/george/work
```

10.6.3 パラメータ・変数展開

変数は { } を使って括るようにすると，間違いがない．

```
$ a="apple"
$ echo a
a
$ echo $a
apple
$ echo ${a}
apple
$ b="banana"
$ echo $ab

$ echo ${a}b
appleb
$ echo ${a}${b}
applebanana
$ echo ${a} ${b}
apple banana
```

変数の文字数を取り出すときは，# を使う．

```
$ a=apple
$ echo ${#a}
5
$ a=9973
```

```
$ echo ${#a}
4
```

開始位置と長さを指定して変数の一部を抽出したり，パターンを指定して変数の一部を削除したり，文字列を置換したりする．

表 9　開始位置と長さを指定して変数の一部を抽出

bash	意味
`${var:i:j}`	開始位置 `i` (先頭からのときは `0`)，長さ `j`
`${var:i}`	開始位置 `i` (先頭からのときは `0`) から最後まで

表 10　パターンを指定して変数の一部を削除

bash	意味
`${var#word}`	先頭から前方最短一致した位置までを取り除く
`${var##word}`	先頭から前方最長一致した位置までを取り除く
`${var%word}`	末尾から後方最短一致した位置までを取り除く
`${var%%word}`	末尾から後方最長一致した位置までを取り除く

表 11　文字列を置換

bash	意味
`${var/word1/word2}`	最初に出てきた `word1` を `word2` に置換
`${var//word1/word2}`	先頭から順番に全ての `word1` を `word2` に置換

ファイルの拡張子を変更したり，ファイルのパス名からファイル名やディレクトリ名を抽出したりすることができる．

コード 40　JPG2jpg.sh

```
1 #!/bin/bash
```

```
2 for file in *.JPG; do
3   mv ${file} ${file//.JPG}.jpg
4 done
```

コード 41　dirbase1.sh

```
1 #!/bin/bash
2 fullpath=/home/george/tako.txt
3 echo 'dirname is'  ${fullpath%/*}
4 echo 'basename is' ${fullpath##*/}
```

ディレクトリ名やファイル名の抽出には，dirname と basename といったコマンドもある．

コード 42　dirbase2.sh

```
1 #!/bin/bash
2 fullpath=/home/george/tako.txt
3 echo 'dirname is'  $( dirname ${fullpath} )
4 echo 'basename is' $( basename ${fullpath} )
```

10.6.4　算術式展開

二重括弧の中は，変数に $ をつけなくてもよい．変数に $ をつけると，算術演算の前に変数展開がおこなわれる．変数に算術式が代入されているとき，変数展開時に算術演算はおこなわれず，変数展開した後に現れる算術式について演算がおこなわれる．

コード 43　arithmatic.sh

```
1 #!/bin/bash
2 a=1+2
3 b=3
4 echo $((   a  *   b  ))
5 echo $((  $a  *  $b  ))
6 echo $(( ($a) * ($b) ))
```

10.6.5　コマンド置換

$(と) に囲まれた部分と，バッククォート ‘ で囲まれた部分は，コマンドとして処理されてコマンドの実行結果がそのまま展開される (ただし末尾の改行は削除される)．最近は，バッククォートを使う方法より，括弧で使う方法が推奨されているらしい．

以下のスクリプトは，filelist.txt に書かれたファイルのバックアップを作成する．

コード 44　bktext.sh

```
1 #!/bin/bash
2 for file in $(cat filelist.txt); do
3   cp ${file} ${file}.bk
4 done
```

10.6.6　パス名展開

ファイル名[53]の指定に使われた * や ? といったワイルドカードが展開される．* や ? はパターンを指定するときに使用する特殊文字で，それぞれ * は長さ 0 文字以上の任意の文字列，? は長さ 1 文字の任意の文字列，である．展開においては，パターンにマッチするファイル名を探し，マッチするすべてのファイル名を並べたリストに置換される．

ワイルドカードには，セット構造と呼ばれるものもある．セット構造は，[と] で挟んで使う．例えば，[abc] は a, b, c のいずれか 1 文字，である．連続するものは最初と最後をハイフン (-) でつないで指定することもできる．例えば，[0-9] は 0 から 9 までの数字，[a-z] は小文字のアルファベット全て，[a-zA-Z] は小文字と大文字のアルファベット全て，を表す．先頭に ! をつけると，除外となる．例えば，[!0-9] は数字以外，である．

シングルクォーテーション (') やダブルクォーテーション (") で囲んだ文字列については，パタンーンマッチから除外され，パス名展開はおこなわれない．

10.6.7　クォートの削除

クォートは削除される．クォートが削除されるということは，Hello と 'Hello' と "Hello" は同じ，ということである．さらに言うと，'H'e'l'l'o' も同じである[54]．

クォート削除の前におこなわれる展開において，シングルクォーテーションは意味をもつことがある．シングルクォーテーション (') で囲んだ文字列では，式が展開されない．

[53] ディレクトリもファイルの一種である．

[54] 意味がないからこんなことは普通の人はやらない．

同じクォートでも，ダブルクォーテーション (") で囲んだ文字列では，式が展開される．

コード 45　quatation.sh

```
1 #!/bin/bash
2 word=apple
3 echo  $word
4 echo '$word'
5 echo "$word"
```

10.7　ファイルからの入力

ファイルの中身を 1 行ずつ読み込む．

コード 46　readfile1.sh

```
1 #!/bin/bash
2 while read line; do
3   echo $line
4 done < $1
```

実行．

```
$ cat foods.txt
apple
banana
carrot
doughnuts
egg
$ ./readfile.sh foods.txt
apple
banana
carrot
doughnuts
egg
```

同じことは，cat とパイプ | を使ってもできる．

コード 47　readfile2.sh

```
1 #!/bin/bash
2 cat $1 | while read line; do
3   echo $line
4 done
```

11　よく使うコマンドたち

ここでは，データを生成したり加工したりするときに使われるいくつかのコマンドについて，その機能の一部を紹介する．

これらのコマンドを覚える必要はない[55]．というか，コマンドは無数にあるので，とても覚えられない[56]．必要なときに必要なものを探し出すことができればよい．最近はネットを使って比較的簡単に検索ができるようになったので，探し方がわかる，ということが重要である．とはいえ，何も知らないと何をどう探したらよいかわからない，というのが普通だろう．覚える必要はないが，見たことはある，くらいの状態になっていないと，探すことはできないだろうし，そもそも探そうと思うことすらないかもしれない．

使い慣れていないコマンドを使うときは，man (A.13 節) を使ってコマンドの機能や構文を確認するとよい．コマンドの多くは，オプションを指定することによって様々な働きをさせることができるので，調べてみるといろいろと便利な機能が見つかるかもしれない．

11.1　grep

文字列を検索する．

[55] ウィザード級の人であっても，覚えているコマンドの数はそんなに多くないと思う．ウィザードとは，基本的なコマンドの組み合わせによって多彩な機能を作り出すことができる人，である (たぶん)．

[56] 研究室のサーバ (epa.desc.okayama-u.ac.jp) で，$ ls /bin /usr/bin | wc -l としてみると，3026 という数字が得られた．重複して数えているものもあるだろうから正確なコマンドの数ではないが，普通の人が覚えられるような量ではない数のコマンドがあることは確かである．

```
$ cat instore.txt
300,apple,2
200,banana,3
50,carrot,5
100,doughnuts,7
150,egg,11
$ grep apple instore.txt
300,apple,2
```

この例では，instore.txt というファイルの中で，apple を含む行を抽出している．

複数のファイルを指定することもできる．

```
$ cat foods.txt
apple
banana
carrot
doughnuts
egg
$ grep apple instore.txt foods.txt
instore.txt:300,apple,2
foods.txt:apple
$ grep apple ./*.txt
instore.txt:300,apple,2
foods.txt:apple
```

2 つ目の例は，カレントディレクトリ (.) にあるファイル名が .txt で終わるすべてのファイルを対象にして検索をおこなったものである．

オプションで -v を指定すると，検索で該当しなかった行を表示する．a という文字を含まない行を表示するなら，

```
$ grep -v a foods.txt
doughnuts
egg
```

11.2 cut

テキストを横方向に分割する．必要な列を選んで抽出することができる．

```
$ cat instore.txt
300,apple,2
200,banana,3
50,carrot,5
100,doughnuts,7
150,egg,11
$ cut -d ',' -f 2 price.txt
apple
banana
carrot
doughnuts
egg
```

区切り文字を `-d ','` で指定．抽出する列は `-f 2` で指定．列は `-f 2,3` や `-f 1-3` のようにして複数指定することも可能．

11.3 sed

文字列を置換したり，行単位で削除したり，抽出したり，いろんなことができる，めちゃくちゃ強力なコマンド．ファイル自体を書き換えるときは，`-i` を付ける．処理の内容は `-e` の後に書く．

置換する (s コマンド).

```
$ sed -e 's/xxx/XXX/g' hoge.txt
```

xxx を XXX で置換する．最後に g を付けると，該当するすべての文字列を置換する．最後に g を付けないと，1 行に 2 つ以上該当する文字列があったとき，1 つ目のみを置換して，2 つ目以降を置換しない．

上の例は区切り記号として / を使っているが，これは別の記号でもよい．パスの置換などで，文字列に / が含まれるときは，| などの記号を使えばよい．

削除する (d コマンド).

```
$ sed -e 2d hoge.txt
```

指定された行を削除する．この例は 2 行目を削除する．5 から 7 行目を削除するなら，5,7d とする．

出力する (p コマンド).

```
$ sed -n -e 2p hoge.txt
```

-n オプションと組み合わせて使用すると，指定した行のみが出力される．

ファイルを書き換える (-i オプション).

```
$ sed -i -e 's/xxx/XXX/g' hoge.txt
```

変更前のファイルを残しておく必要があるときは，-i を付けてはならない．ファイルに

書き出す必要があるなら，リダイレクト > を使う．

特定の文字列で囲まれた範囲を抽出する．

```
$ sed -n -e '/^BEGIN$/,/^END$/p' hoge.txt
```

これは BEGIN のみの行から END のみの行までを表示する．

11.4 awk

複数列のデータを処理する．
基本的な使い方は，

```
$ awk ' パターン{アクション}' ファイル名
```

1 行ずつ読み込んで，パターンに合致したら，アクションで指定された内容を実行する．パターンを指定しないと，全ての行が対象となる．アクションを省略すると，行全体を出力する．

列を抽出する．

```
$ awk '{print $1}' hoge.txt
```

出力するときは print を使う．列は $ の後ろに数字を書いて指定する．列として 0 番目を指定すると，行全体を指定したことになる．

```
$ awk '{print $0}' hoge.txt
```

これは

```
$ cat hoge.txt
```

と同じである．

区切り文字は -F で指定する．CSV (Comma-Separated Values) を処理するときは，-F, とすればよい．

```
$ awk -F, '{print $1 "," $2}' hoge.csv
```

何も指定しないときは，1 つ以上のスペースやタブが，区切り文字となる．連続しているスペースとタブは，その数によらず，ひとつの区切りと認識される．

複数の列を指定することもできる．

```
$ awk '{print $1 $2}' hoge.txt
```

これだと，1 列目と 2 列目がくっついて出力される．区切り文字を入れるときは，":" などとする．

```
$ awk '{print $1 ":" $2}' hoge.txt
```

出力を CSV にするなら，"," とすればよい．

特定の列が特定の文字列になっている行を抽出する．

```
$ awk '$2=="apple"' hoge.txt
```

これは 2 列目が apple の行を抽出する.

特定の文字列になっている行を除外するときは，!= を使う.

```
$ awk '$2!="banana"' hoge.txt
```

これは 2 列目が banana の行を削除する.

現在処理している行番号は NR で参照することができる.

```
$ awk '{print NR " " $0}' hoge.txt
```

これは左端に行番号を追加する．コードを出力するときに重宝する.

出力には sprintf を使うことができる.

```
$ awk '{print sprintf("%04d",NR) " " $0}' hoge.txt
```

長いコードを出力するときは，こういう工夫をしておかないと始まりの位置がずれてしまう.

11.5 sort

行を並べ替えるコマンド.

```
$ cat foods1.txt
coconut
 carrot
apple
banana
apricot
$ sort foods1.txt
 carrot
apple
apricot
banana
coconut
```

文字として左側から昇順に並べ替えをおこなう．標準では行頭の空白を無視しない．

行頭の空白を無視するときは，-b を追加する．

```
$ sort -b foods1.txt
apple
apricot
banana
 carrot
coconut
```

出力は空白を保持する．

数字の並べ替えはちょっと注意が必要．

```
$ cat price.txt
300
```

```
200
50
100
150
$ sort price.txt
100
150
200
300
50
```

標準では，文字として左側から昇順に並べ替えをおこなう．そのため，50 は数字としては一番小さいにも関わらず一番最後になっている．

数値として並べ替えるときは，オプションで -n を追加する．

```
$ sort -n price.txt
50
100
150
200
300
```

- の符号は認識するが，+ は認識できない．また，指数表記も認識しない．

降順で並べ替えるときは，オプションで -r を追加する．

```
$ sort -n -r price.txt
300
200
150
```

```
100
50
```

11.6 uniq

uniq コマンドで処理するデータは，あらかじめ並べ替えされている必要がある．並べ替えは sort コマンド (11.5 節) を使えばよい．

uniq コマンドで重複する行を削除する．

```
$ cat foods2.txt
apple
banana
banana
carrot
doughnuts
egg
egg
egg
$ uniq foods2.txt
apple
banana
carrot
doughnuts
egg
```

各行が何行ずつあったか表示するときは，-c を使う．

```
$ uniq -c foods2.txt
   1 apple
   2 banana
   1 carrot
   1 doughnuts
   3 egg
```

11.7 xargs

xargs は，標準入力から受け取ったデータをパラメータにしてコマンドを実行する．
ファイルリストにあるファイルを削除する．

```
$ cat filelist.txt
hoge.txt
hero.txt
$ xargs rm < filelist.txt
```

この例では hoge.txt と hero.txt が削除される．

カレントディレクトリの下にあるすべての .DS_Store を削除する．

```
$ find . -name '.DS_Store' | xargs rm
```

Mac ユーザの需要に答えるコマンド．

コマンドに渡す引数の数の最大は，-n を使って指定することができる．

```
$ cat /etc/hosts.deny | grep -v # | awk '{print $2}' | xargs -n 1 host
```

引数がたくさんあるとき，n で指定された数ずつ，引数がコマンドに渡されて処理がおこなわれる．コマンドによっては，一度に処理できる引数の数に制限があったりする．また，負荷の大きい処理は分散させて実行するのがよい．

-I オプションを使って文字列を指定すると，コマンド実行時に指定した文字列が標準入力から受け取ったデータに置換される．

```
$ ls *.txt | xargs -I FILE cp FILE FILE.bk
```

まず，カレントディレクトリ内にあるファイルのうちファイル名が .txt で終わるファイルのリストが作られ，それがパイプで xargs に渡される．xargs は「cp FILE FILE.bk」を実行するとき，FILE をパイプで送られてきたファイル名に置き換えて実行する．

-P オプションを使って，並列化することができる．

```
$ ls *.txt | xargs -P 8 -I FILE cp FILE FILE.bk
```

11.8 paste

paste はファイルを水平結合する．

```
$ paste price.txt foods.txt
300     apple
200     banana
50      carrot
```

```
100     doughnuts
150     egg
```

結合するファイルは 3 つ以上も可.

```
$ paste price.txt foods.txt stock.txt
300     apple   2
200     banana  3
50      carrot  5
100     doughnuts       7
150     egg     11
```

区切り文字も指定可能．オプションで区切り文字を指定しないとき，区切り文字はタブになる．区切り文字として「,」を指定すれば，CSV (Comma-Separated Values) も簡単に作ることができる.

```
$ paste -d , price.txt foods.txt stock.txt
300,apple,2
200,banana,3
50,carrot,5
100,doughnuts,7
150,egg,11
```

ちなみに，ファイルを垂直結合するときは `cat` を使えばよい.

```
$ cat price.txt foods.txt
300
200
```

```
50
100
150
apple
banana
carrot
doughnuts
egg
```

11.9 wc

行数を数える．

```
$ wc -l hoge.txt
```

行数は改行の数を数えている．最後に改行のない行があると，その行はカウントされない．改行が 1 つもないときは，0 になる．

ディレクトリにあるファイルの数は以下で数えることができる．

```
$ ls | wc -l
```

11.10 diff

ファイルを比較して，差異があれば表示する．

```
$ diff file1.txt file2.txt
```

12　シェルスクリプトの演習問題

12.1　ログの解析

12.1.1　ウェブアクセスの解析

12.1.2　ログインログの解析

12.2　アメダス

12.2.1　データのダウンロード

12.2.2　データの整形

12.2.3　データの可視化

第 VI 部

画像処理

13　ImageMagick

ImageMagick のコマンドは，安直に画像処理をするときに便利に使える．
ImageMagick コマンドリファレンス http://image-magick.com/

13.1　気象衛星ひまわり

14 ImageJ

14.1 岡山の空

付録 A　Unix の使い方

Unix コマンドはたくさんあるが，頻繁に使うものはそんなに多くない．頻繁に使ういくつかのコマンドを覚えれば，だいたいの用は足りる．多くのコマンドは，引数と組み合わせて使う．引数とは，コマンドとセットにして入力する，コマンド以外の文字のことである．

コマンド入力の約束．

- 大文字と小文字を区別する
- コマンド (と引数) を入力したら最後に『エンター』キーを押す
- コマンドと引数の間には空白文字を入れる
- 引数と引数の間にも空白文字を入れる

A.1　ls : ファイル名の表示

ファイルの名前は

```
$ ls
```

とすると，表示される．`-l` オプションを付けると，時刻やサイズを含んだ情報が表示される．

```
$ ls -l
```

．で始まる名前のファイルは隠しファイルと呼ばれ，普通に `ls` としただけでは表示されないようになっている．隠しファイルを表示したいときは，`-a` オプションを付ける．

```
$ ls -a
```

A.2　mv：ファイル名の変更

ファイル名を old から new に変更する.

```
$ mv old new
```

コマンド実行前に new が存在した場合，コマンドの実行によって実行前に存在した new は消去される.

A.3　cp：ファイルの複製

file1 と同じ中身を持つ file2 を作成する.

```
$ cp file1 file2
```

コマンド実行前に file2 が存在した場合，コマンドの実行によって実行前に存在した file2 は消去される.

ディレクトリ丸ごと複製をつくるときは，-r のオプションを付ける (それ以外はファイルの複製をつくるときと同じ).

```
$ cp -r dir1 dir2
```

A.4　rm：ファイルの消去

ファイルの消去するときは,

```
$ rm filename
```

A.5　cat：テキストファイルの中身を見る

テキストファイルの中身を表示するには

```
$ cat filename
```

バイナリ[57]ファイルを cat で表示してはいけない (たいへんなことになります).

A.6　more：テキストファイルの中身を見る

端末の 1 画面に収まり切らないときは，more を使う.

```
$ cat filename
```

『スペース』キーを押すと 1 ページ進む，『B』のキーを押すと 1 ページ戻る，『Q』のキーを押すと終了.

A.7　pwd：現在地の表示

現在地の表示.

```
$ pwd
```

[57] バイナリ (binary) は文脈によって指すものが変わる，面倒な単語である．計算情報関連の世界では “機械が読むことのできるデータ” という意味で使われる．計算機が扱うデータはすべて定義によってバイナリということになる．バイナリ (＝計算機が扱うすべてのデータ) のうち，人間が読むことのできる文字や記号で表された一部の形式のデータは “text 形式” と呼ばれる．計算機が扱うすべてのデータから text 形式のデータを除いたものに限定して，バイナリという言葉を使うことがある．このとき，バイナリはそのままでは人間が読むことのできない形式のデータ，ということになる．さらに範囲を限定して，実行可能形式のみを指してバイナリと呼ぶこともある.

A.8　mkdir：ディレクトリの作成

ディレクトリの作成する．

```
$ mkdir dirname
```

A.9　rmdir：ディレクトリの削除

ディレクトリを削除する．

```
$ rmdir dirname
```

削除しようとしているディレクトリにファイルがあるとき，ディレクトリを削除することはできない．ディレクトリを削除する前に，ディレクトリ内にあるファイルを削除しておく．

ディレクトリの削除は，rm に -r のオプションを付けることによっても可能である．

```
$ rm -r dirname
```

このコマンドを実行すると，指定したディレクトリとその下にある全てのファイル (ディレクトリを含む) が削除される．

A.10　cd：ディレクトリの移動

ディレクトリの移動．移動先のディレクトリを指定して，移動する．

```
$ cd dirname
```

ひとつ上のディレクトリに移動するときは,

```
$ cd ..
```

A.11　chmod：パーミッションの変更

パーミッションの変更.

A.12　passwd：パスワードの変更

パスワードの変更.

A.13　man：コマンドのマニュアルを表示する

コマンドの動作や引数の指定方法について知りたいときに使う．例えば，`ls` について知りたいときは,

```
$ man ls
```

マニュアルは `more` を使って表示される (終了するときは『Q』のキーを押す).

A.14　ワイルドカード

`*` の使い方.

A.15　ディレクトリの階層構造

階層構造の説明.
相対パスと絶対パスの説明.

A.16　参考書

とりあえず，自習用の書籍として [4] をあげておくが，ネット上を探せば書籍以外にもいろいろ見つかると思う．

[5] と [6] は辞書のように使う本として，本棚に入れておく本であったが，絶版になっている．この本が絶版になった後，これに代わる本があるのかないのか知らない．最近は紙の本は持たず，ネット上で検索して済ませる人が多いかもしれない．

付録 B　ネットワーク

B.1　リモートログイン

ssh の使い方

B.2　ファイル転送

scp の使い方

謝辞

コード 3 の解説の第 1 稿は村上真也氏が書いたものである．その他，村上氏からもらったいくつかのコメントに基づいて文書を改訂した．ありがとうございました．

参考文献

[1] Dehnadi, S., and R. Bornat (2006) The camel has two humps. `http://www.eis.mdx.ac.uk/research/PhDArea/saeed/paper1.pdf`

[2] Bornat, R. (2014) Camels and humps: a retraction. `http://www.eis.mdx.ac.uk/staffpages/r_bornat/papers/camel_hump_retraction.pdf`

[3] ブライアン・カーニハン (著)，ロブ・パイク (著)，福崎俊博 (訳) (2000)「プログラミング作法」アスキー，pp.355.

[4] 大津真 (2015)「6 日間で楽しく学ぶ Linux コマンドライン入門」インプレス R&D，pp.333.

[5] 山口和紀，古瀬一隆 (2003)「新 The UNIX Super Text 上 改訂増補版」技術評論社，pp.870.

[6] 山口和紀，古瀬一隆 (2003)「新 The UNIX Super Text 下 改訂増補版」技術評論社，pp.1014.