Mathematica ドリル (1991年版)

Nobuki Takayama

1991年

目 次

まえがき

1991 年版そのままです. 実行環境は NeXT コンピュータです. 今の MacOS X はけっこうこれがベースになったりします. 昔の Mathematica なので, graphic の扱いに微妙な違いがあります. このあたりはそのうちに補足説明を書きます. また, このころは screenshot を取るのも大変だったので, 図がはいっていなかったりします. Risa/Asir ドリル はこの Mathematica ドリルを元に作成されました. 逆に Risa/Asir を使える人が Mathematica を覚えるのに参考になると思います.

2012.06.14, 著者

第1章　NeXT の操作

1.1　login, logout, ディレクトリブラウザ

1. 電源スイッチはキーボードにある。図 1.1 をみよ。もし電源がはいっていなかったら、power と書いてあるキーをおすと NeXT が動く。電源をいれてから、使用可能な状態になり図 1.2 の login ウインドウーが現れるまで数分かかる。電源断はしなくてよい。もしするときは Network から使用している人がいないか調べてからやる。

2. 図 1.1 をみよ。おひさまマークのキーで画面を明るくしたり暗くしたりする。スピーカーマークのキーで音を大きくしたり小さくしたりする。

3. キーはちょんとおせばよい。おしつずけるとその文字がいっぱい入力される。

4. COMMAND キーや SHIFT キーや CTRL キーは他のキーと一緒に押すことで始めて機能するキーである。これらだけを単独に押してもなにもおきない。以後 SHIFT キーをおしながら他のキーを押す操作を SHIFT+きー と書くことにする。 command キー、 ctrl キーについても同様である。

5. SHIFT+a とすると大文字の A を入力できる。

6. BackSpace を押すと一文字前を消去できる。NeXT 以外の計算機ではこのキーは BS とか DEL と書いてあることが多い。

7. SPACE キーは空白を入力するキーである。空白も一つの文字である。(アスキーコード 20H)

8. ’ と ‘ は別の文字である。注意すること。また、プログラムリストを読む時は 0 (ゼロ）と o （おー）の違いにも注意。

9. login (利用者の認証）を行なわないと利用の開始ができない。(図 1.2)

10. logout で使用の終了を宣言する。

11. マウスの操作には次の三種類がある。

    (a) クリック: 選択するとき、文字を入力する位置（キャレットの位置）の移動に用いる。マウスのボタンをちょんとおす。

    (b) ドラッグ: 移動、サイズの変更、範囲の指定、プルダウンメニューをひっぱるときなどに用いる。マウスのボタンを押しながら動かす。

    (c) ダブルクリック：プログラムの実行、open(ファイルを開く) をするために用いる。マウスのボタンを 2 回つずけてちょんちょんとおす。ダブルクリックをしたアイコンは白くなる。白くなったら待つ。混んでいるときは起動に 1 分以上の時間がかかることもあり。むやみにダブルクリックを繰り返すとその回数だけ起動されてなお遅くなる。

図 1.1: キーボード

図 1.2: login と logout

図 1.3: アプリケーションドッグ

問題 1.1 [01]　login して logout してみる– 文字の入力とクリック。

問題 1.2 [01]　ディレクトリブラウザの大きさを変えてみる– ドラッグ。

問題 1.3 [01]　ディレクトリブラウザーを用いて`/NeXTDeveloper/Demos` まで移動し `Billiards` とか `Synthesizer` を実行してみる– ダブルクリック。

問題 1.4 [01]　`/NeXTLibrary` へ移動してマニュアル（使用説明書）を読んでみよう（英語です）。

1.2　Mathematica の使い方

1. Mathematica を起動するにはアプリケーションドッグ (図 1.3) にある Mathematica のアイコンをクリックするか、ディレクトリブラウザー (図 1.4) で `/NeXTApp` まで移動して Mathematica.app を起動する。終了は Mathematica の メインメニューの `Quit` をクリックする。
2. 大文字、子文字を区別しているので注意。システムに組み込まれた関数、定数は大文字で始まる。
3. `+,-,*,/,^` は足し算、引き算、かけ算、割算、巾乗。
4. `x2` と書くと、 $x2$ という名前の変数となる。x かける 2 は `x*2` と書く。
5. 数学ではかっことして、`[,]`,`{,}` などがつかえるが Mathematica では `(,)` のみ。`[,]` や `{,}` は別の意味である。
6. `Expand[` *poly* `]` は *poly* を展開する。
7. `Factor[` *poly* `]` は *poly* を有理数係数の多項式環で因数分解する。

図 1.4: ディレクトリブラウザー

8. `Sin[x], Cos[x]` はおなじみの三角関数。`Pi` は円周率を表す定数。

9. `Plot[` f `, {` x, $xmin$, $xmax$ `} ]` は x の関数 f のグラフを $x = xmin$ から $x = xmax$ まで描く。

10. `For[k=初期値, k<=終りの値,k++,` ループの中で実行するコマンド`]` はあることを何度も繰り返したい時に用いる。for ループと呼ぶことにする。

11. `Print[` *expr* `]` は *expr* を出力する。

12. window のサブメニューの open, save, save as

13. print, print selection

14. 実行中のものを中断したい時はメインメニューのなかの `Action` を押したままにするとサブメニューがでるので、`Interrupt` までドラッグしていってから離す (図 1.5)。もし `Action` メニューがすでに出ていれば、`Interrupt` をクリックする。すると dialog box がでるか、

```
Interrupt:
        continue (or c) to continue
        show (or s) to show current operation (and then continue)
        trace (or t) to show all operations
        abort (or a) to abort current calculation
        exit (or quit) to exit program
?
```

と表示されるので a `RETURN` を入力する。

図 1.5: Interrupt

例題 1.1 [02] `Expand[(x+y)^ 3]` を実行してみよう。入力してから

COMMAND + RETURN

を押すと実行する。

入力例 1.1

```
In[1]:= Expand[(x+y)^3]
         3      2          2    3
Out[1]= x  + 3 x  y + 3 x y  + y

In[2]:=
```

例題 1.2 [02] 上の入力をエディットして変更することにより、

`Expand[(x+z)^ 5]`

を実行してみる。

入力例 1.2 略

変数に値をとっておくこともできる。

例題 1.3 [02] 次の命令を実行してみよう。

```
kazu = 5
hehehe = x+ 8
kazu+ hehehe
Clear[kazu]
kazu
```

入力例 1.3

```
In[1]:=  kazu = 5
Out[1]= 5
In[2]:=  hehehe = x+ 8
Out[2]= 8 + x
In[3]:=  kazu+ hehehe
Out[3]= 13 + x
In[4]:=  Clear[kazu]
In[5]:=  kazu
Out[5]= kazu
In[6]:=
```

例題 1.4 [02] `Factor[x^ 5 - y^ 5]` を実行してみなさい。

入力例 1.4

```
In[6]:= Factor[x^5-y^5]
                   4    3      2  2      3    4
Out[6]= (x - y) (x  + x  y + x  y  + x y  + y )
```

図 1.6: Mathematica のフロントエンド

例題 1.5 [02] (図 1.6)　`Plot[Sin[x], x,0,2*Pi]` を実行してみなさい。[1] もし実行したら運悪く

```
Show::nomed: No graphics output medium specified.
Out[7]= -Graphics-
```

というエラーメッセージがでたら、`<<Terminal.m` と入力する。もし Tektronics グラフィックターミナルのエミュレータを使っているのなら, `<<Tek.m` と入力する。

入力例 1.5 NeXT の画面にはきれいな絵を描くことができるが、絵を描くことができない文字ターミナルでも、図 1.7 のように絵を書くことができる。

例題 1.6 [02]　上の例で `Sin` を `Tan` に変えて実行してみなさい。

入力例 1.6 略

例題 1.7 [02]

```
Plot3D[Sin[x]*Cos[y], {x,0,2*Pi},{y,0,2*Pi}]
```

を実行してみなさい。`Plot3D` はグラフを三次元表示する。

入力例 1.7　図 1.8 を見よ。

例題 1.8 [02]　`?Plot` と入力すると関数 `Plot` の使い方の説明が出る。また、`?Pl*` と入力すると `Pl` ではじまる組み込み関数のリストがでる。`*` は任意の文字列をあらわす。

入力例 1.8

```
In[11]:= ?Plot
Plot[f, {x, xmin, xmax}] generates a plot of f as a function of x from xmin to
   xmax. Plot[{f1, f2, ...}, {x, xmin, xmax}] plots several functions fi.

In[11]:= ?Pl*
Plain        Plot3D       PlotColor    PlotJoined   PlotPoints   PlotStyle
Plot         Plot3Matrix  PlotDivision PlotLabel    PlotRange    Plus
```

例題 1.9 [02]

```
For[k=1,k<=5,k++, Print[k]]
```

を実行してみなさい。このプログラムは `Print` を k の値を 1 から 5 まで変えながら 5 回実行する。

入力例 1.9

```
In[10]:= For[k=1,k<=5,k++,Print[k]]
1
2
3
4
5
```

[1] 実習室の genmai1,.., sencha1,..., hojicha1,... では機械の能力不足のためグラフィック関係のコマンドは実行できないことが多い。ほかにだれもほとんど使っていないなら sencha, hojicha などのサーバマシンでならなんとかやれる。

```
In[8]:= <<Terminal.m
 -- Terminal graphics initialized --

In[9]:= Plot[Sin[x],{x,0,2*Pi}]
```

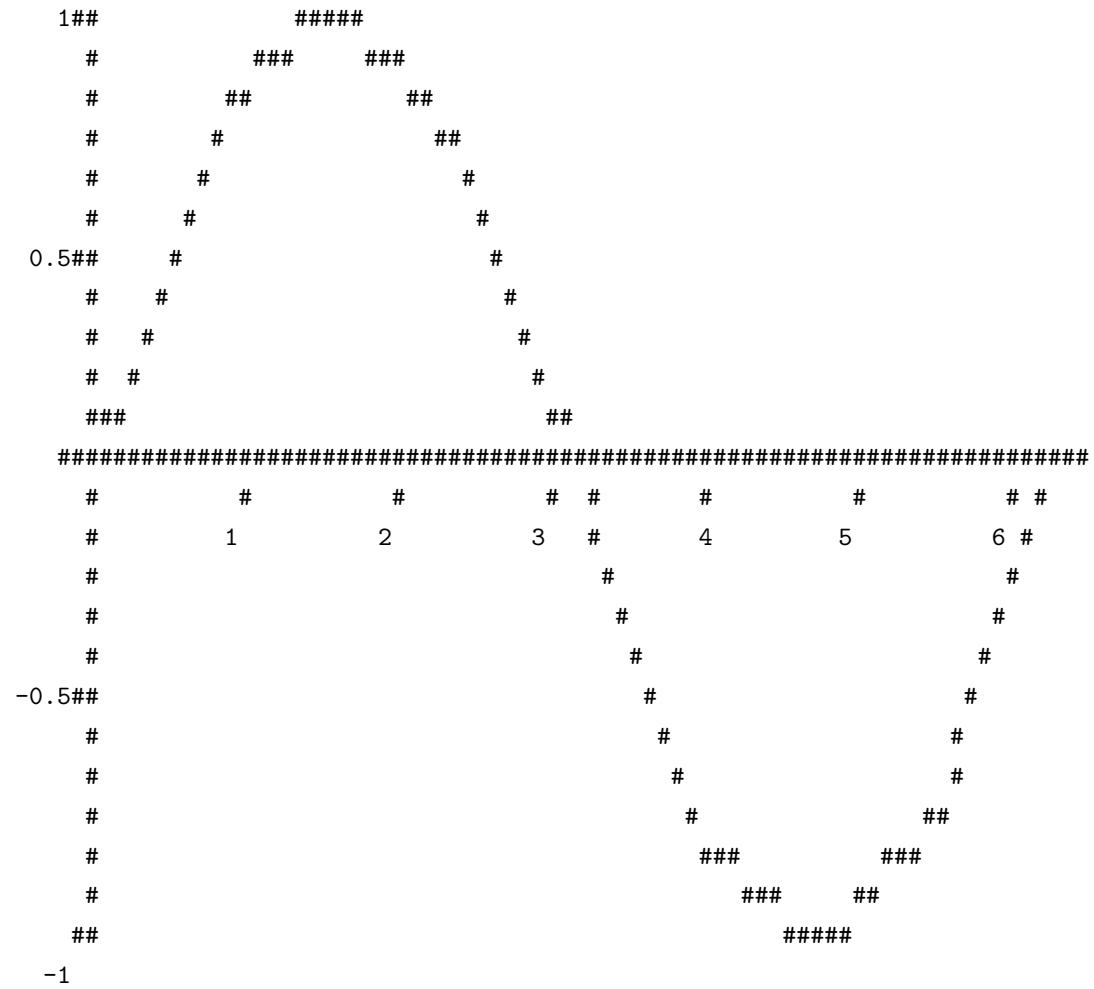


図 1.7: Terminal.m によるグラフ

図 1.8: NeXT の window system でのグラフィックス

図 1.9: プリント

例題 1.10 [02]　１から１００までの数の和を求めるプログラム

```
re = 0;
For[k=1, k<=100, k++,
    re = re+k];
Print[re];
```

または

```
main[]:=Block[{re,k},
        re = 0;
        For[k=1, k<=100, k++,
            re = re+k];
        Print[re]; ];
main[];
```

を実行してみなさい。

入力例 1.10 略

例題 1.11 [02]　例題の 1.7 の出力結果をプリンターへだしてみなさい。マウスで図の右側の縦線のあたりをクリックしてプリントするものを選択してから[2] メニューの `Print Selection` をクリックすればよい。図 1.9 を見よ。

入力例 1.11 略

例題 1.12 [02]　いままでのノートブックの内容をメインメニューの Window サブメニューの `Save As..` で名前/好きな名前にセーブ (save) し logout しなさい (たとえば takayama/test1 など。 図 1.10)。次回からはメインメニューの Window サブメニューの `open` を実行することでこのノートブックの内容を読んでくることができる。

入力例 1.12 図 *1.10* を見よ。

[2] 2 個以上のセルを選択するには `shift` をおしながらクリックしていけばよい。

図 1.10: save

関連図書

[1] The NeXT book, B.F. ウェブスター著、アジソンウェスレー、トッパン (日本語です)、３８００円。NeXT の使い方を解説した本。

[2] S. Wolfram, Mathematica , A system for Doing Mathematics by Computer, Addison-Wesley, (英語、Paperback は５千円以下の値段だったと記憶する。)。これは Mathematica のマニュアルであり、最良の参考書である。

[3] 小池慎一、Mathematica 数式処理入門、技術評論社、 2500 円。これも良い本。Macintosh 版の Mathematica の使い方と Mathematica の各関数の使い方を説明してある。

第2章　unix のシェルとエディターemacs

コンピューターが利用者（ユーザー）とやりとりをするには2通りのやり方がある。一つが graphical user interface でありもうひとつが文字ベースの user interface である。核となるシステムがありそれを覆っているのが user interface である。NeXT は一章でみたような graphical user interface とこの章でみるような文字ベースの user interface 両方を持っている。NeXT の場合、核となっているのは unix オペレーティングシステム[1] である。

2.1　unixのシェルコマンド

1. 入力要求記号 (プロンプト) はコンピュータが何か入力を求めている時に表示される。入力要求記号は文字ベースのユーザインターフェースで良く使われる。入力要求記号は動いているソフトウエアの種類によってことなる。たとえば unix のシェルは `%` とか `$` とか `hojicha2>` などを入力要求記号として使っている。Mathmatica は `In[3]:=` とかを入力要求記号として使っている。MS-DOS のシェル (command.com) は `A>` とかを入力要求記号として使っている。以下でシェルに対するコマンドを説明する。

2. NeXT 本体でシェルを立ちあげたい時は、ディレクトリブラウザー (図 1.4) で、`/NeXTApp` までいき、`Shell` か `Terminal` を立ちあげる。ウインドーが開き、unix シェルのプロンプトがでる。

3. `telnet` ホストの名前：そのホストへリモートログインする。失敗すると `telnet>` と表示されるので `quit` と入力すればシェルにもどれる。

4. `math` RETURN：Mathematica を起動する。`Quit` RETURN：Mathematica を終了する。Mathematica のある関数を実行中に中断したいときは CTRL+C を押す。(Interrupt と同じこと)

5. `rwho`：いまネットワーク上にだれが login しているかをみるコマンド。

6. `ls`：カレントディレクトリのファイルの一覧を表示する。`ls -l`：カレントディレクトリのファイルの一覧を詳しく表示する。`ls -l | more` とすれば一ページずつ表示できる。`ls -a`：`.` ではじまる隠されたファイルも表示する。

7. `mkdir` ディレクトリ名：新しいディレクトリを作る。`cd` ディレクトリ名：そのディレクトリにカレントディレクトリを移す。`cd ..`：一つ上のディレクトリへ移る。`pwd`：カレントディレクトリ (現在いるディレクトリ) を表示する。

8. `chmod`：ファイルの属性をかえる。`chmod -w`　ファイル名：ファイルを書き込み禁止にする。`chmod 600`　ファイル名：ファイルを自分だけしか読み書きできないようにする。

[1] 実は mach operating system がうごいている。mach は unix に上位互換な機能を持っている。

9. more

10. cp　ファイル名 1　　ファイル名 2 : ファイルのコピーをする。rm　ファイル名 : ファイルを消す。

11. ps -ax : 現在動いているプロセスをすべて表示する。kill -9 プロセス番号 : プロセスを殺す。

12. CTRL+C : プログラムを stop する。

13. CTRL+Z : プログラムを中断する。kill %% : いま中断したプログラムをころす。fg : いま中断したプログラムをぞっこうする (forground にもってくる)。

14. ftp : ethernet でつながった機械の間でのファイル転送。

15. passwd : パスワードの変更。

2.2　emacs

1. emacs ファイル名 : emacs の起動。 emacs でなく nemacs と入力するときもある。

2. CTRL+x　CTRL+f　ファイル名　RETURN : ファイルの読み込み。

3. CTRL+x　CTRL+c : emacs の終了。

4. CTRL+x　CTRL+s : save 。 ファイルをセーブする。

5. 以下がカーソルの移動コマンド。CTRL+f : Forward 。CTRL+b : Backword 。CTRL+p : Previous 。 CTRL+n : Next 。

6. CTRL+d : delete 。カーソルの上の一文字をけす。 CTRL+k : カーソルより行末までの文字を消す。

7. CTRL+g : 中断。これはとても便利。

8. CTRL+j : 一行したへ移動する。言語に応じたインデントを自動的にしてくれる。

9. CTRL+SPACE : マークをつける。 esc w : write to yank buffer。バッファにかく (copy)。CTRL+y : yank 。バッファの中身をカーソルのいちに書く (paste)。 CTRL+w : マークをつけた位置からカーソルまでの領域をバッファへ移す (copy)。

10. CTRL+_ : Undo。つまり、” 待った！”。

11. CTRL+x 2 : Window を二つにする。CTRL+x 1 : Window を一つにする。CTRL+x o : other window。他のウインドーへ移る。

12. esc x shell RETURN : emacs のなかからシェルを立ちあげる。

例題 2.1 [02]　PC9801 より telnet を用いて NeXT (たとえば sencha1, sencha2 ,...) へリモートログインしてみなさい。

入力例 2.1

```
A>telnet sencha
Trying 133.30.11.3 ...
Connected to sencha.
Escape character is '^]'.

4.3 BSD UNIX (sencha)

login: guest
Password:guest
Last login: Mon Mar 11 14:57:30 from 133.30.68.1
% Warning: no termcap entry for kterm.  Editing disabled.

% math
Mathematica 1.2 (August 30, 1989) [With pre-loaded data]
by S. Wolfram, D. Grayson, R. Maeder, H. Cejtin,
   S. Omohundro, D. Ballman and J. Keiper
with I. Rivin and D. Withoff
Copyright 1988,1989 Wolfram Research Inc.

In[1]:= Factor[x^2-y^2]

Out[1]= (x - y) (x + y)

In[2]:= Quit
% logout
Connection closed by foreign host.
A>
```

または

```
A>telnet
telnet> open sencha              (* sencha を呼び出す。 *)
Trying 133.30.11.3 ...
Connected to sencha.
Escape character is '^]'.

4.3 BSD UNIX (sencha)

login: guest
Password:
Last login: Sat Apr 13 10:18:55 from gradura
% Warning: no termcap entry for kterm.  Editing disabled.

% setenv TERM vt100              (* unsetenv TERMCAP がいる時もある。 *)
% math
```

```
Mathematica 1.2 (August 30, 1989) [With pre-loaded data]
by .... 以下省略
```

例題 2.2 [02]　1 から 100 までの和を求めるプログラムを s1.m という名前で emacs を用いて作りなさい。そしてこれを mathematica へ読み込んで実行しなさい。In[1]:= <<s1.m と入力すれば mathematica へ s1.m をよみこむことができる。

s1.m

```
re = 0;
For[k=1, k<=100, k++,
   re = re + k
   ];
Print[re];
```

入力例 2.2

```
gradext3> mkdir takayama          自分用のディレクトリを作る。
gradext3> cd takayama             そのディレクトリへ移る。
gradext3> emacs s1.m              そこで s1.m という名前でファイルを編集
            <<< 編集する>>> <<< emacs 終了 >>>
gradext3> math                    mathematica の起動
Mathematica 1.2 (August 30, 1989) [With pre-loaded data]
by S. Wolfram, D. Grayson, R. Maeder, H. Cejtin,
   S. Omohundro, D. Ballman and J. Keiper
with I. Rivin and D. Withoff
Copyright 1988,1989 Wolfram Research Inc.

In[1]:= <<s1.m                     ファイル s1.m の読み込み
5050                               Print[re] が答を表示した。

In[2]:=
```

この例のようにやると emacs と Mathematica の間をいったりきたりしなくてはならない。立ちあげるのに結構時間がかかって能率が悪い。このようなときには、 NeXT の上なら Terminal を 2 つ以上立ちあげると都合がよい。また、PC9801 よりリモートログインして NeXT を使っているなら、emacs を立ちあげてそのなかから Mathematica を呼んでみるのも能率がよい。[2]

例題 2.3 [02]　NeXT の上でターミナルエミュレータ (Terminal) をふたつ動かして s1.m を動かしてみなさい。

入力例 2.3　図 2.1 を見よ。

図 2.1: Terminal

例題 2.4 [02]　emacs のウインドーを二つに分割して (esc x 2), 一方のウインドーで shell をたちあげ (esc x shell RETURN), mathematica を動かしなさい。

入力例 2.4　図 2.2 を見よ。上のウインドーが shell ウインドー , 下が s1.m を編集しているウインドー である。

このように、一つの画面を二つに分割して計算機を使用することができるが、いっぱいコンピュータがならんでいたら、一人で 2 台使うというのが一つの賢いやりかたである。

例題 2.5 [03]　三章以降のプログラム例を emacs で入力して実行してみなさい。

入力例 2.5 略

以下は nemacs のヘルプファイルのハードコピーである。

==========
日本語 GNUEMACS(NEmacs) 入門編
==========

注意：　この入門編は、「習うより慣れろ」をモットーに作成されています。行の左端の ">>" は、その時何をすべきかを指示しています。

NEmacs のコマンドを入力するときには、一般的にコントロール・キー（キー・トップに、CTRL あるいは、CTL と書いてある）やメタ・キー（普通、エスケープ・キーを使う）が使われます。そこで、CONTROL とか META とか書く代わりに、次のような記号を使うことにします。

C-<文字>　コントロール・キーを押したまま、<文字>キーを押します。例えば、C-f　は、コントロール・キーを押しながら f のキーを押すことを意味します。

>> それでは、C-v（View Next Screen; 次の画面を見る）をタイプしてみて下さい。次の画面に進むことができます。

ESC <文字>　エスケープ・キーを押してから離し、それから<文字>キーを押します。

注意：　<文字>は、大文字でも小文字でもコマンドとしては同じ意味になります。

重要です：　Emacs を終了させたい時は、C-x C-c をタイプします。Emacs を csh

[2] Emacs の math.el を使うという方法もある。 math.el は emacs 用の mathematica フロントエンドである。

図 2.2: emacs の中からシェルを立ちあげる。

から起動している場合、サスペンドする（一時的に止める）ことが
出来ます。Emacs をサスペンドするには、C-z をタイプします。

さて、これからは、一画面分読み終えたら、C-v　を入力して行って下さい。

前の画面と次の画面とでは、表示される内容に何行文かの重なりがあります。これは、表示されている内容が連続していることがすぐ判るようにするためです。

まずは、ファイルの中を移動して行く方法を知る必要があります。C-v　によって先に進むことはもう判りました。元に戻るには、ESC v をタイプします。

>> ESC v と C-v　を使って、前後に移動することを何回か試してみなさい。

要約
====

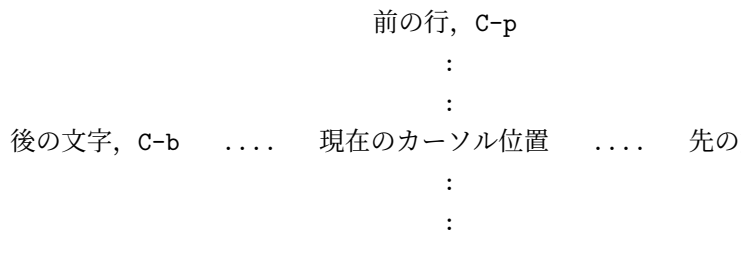
ファイルを画面毎に見て行くには、次のコマンドを使います。

C-v	前に一画面分進む
ESC v	後ろに一画面分戻る
C-l	画面を書き直す。このとき、元カーソルのあった行が画面の中央にくるようにする

>> 今カーソルがどこにあるか、その近くにどんなテキストが書かれているかを覚えなさい。C-l　をタイプし、カーソルがどこに移動したか、その近くのテキストはどうなったかを調べてみなさい。

基本的なカーソルの制御
======================

画面毎の移動はできるようになりました。今度は、画面の中で、特定の場所に移動するための方法を憶えましょう。これにはいくつかのやり方があります。一つの方法は、前 (previous) 次 (next) 先 (forward) 後 (backward) に移動するコマンドを使うことです。これらのコマンドはそれぞれ、C-p, C-n, C-f,C-b に割り当てられており、現在の場所から新しい場所にカーソルを移動させます。図で書けば、



次の行, C-n

これらは、それぞれ、Previous, Next, Backward, Forward の頭文字になっているので、憶えやすいでしょう。これらは、基本的なカーソル移動コマンドであり、いつでも使うものです。

>> C-n を何回かタイプし、（今、あなたが読んでいる）この行までカーソルを移動させなさい。

>> C-f を使って行の中ほどに移動し、C-p で何行か上に移動してみなさい。カーソルの位置の変化に注意しなさい。

>> 行の先頭で C-b をタイプしてみなさい。カーソルはどこに移動しますか？さらにもう少し C-b をタイプし、今度は C-f で行末の方に戻りなさい。カーソルが行末を越えるとどうなりますか？

画面の先頭や末尾を越えてカーソルを移動させようとすると、その方向にあるテキストが移動して来て、カーソルは常に画面内にあるようにされます。

>> C-n を使って、カーソルを画面の下端より下に移動させてみなさい。何が起こりましたか？カーソルの位置はどう変りましたか？

一文字単位の移動ではまどろっこしいと思うなら、単語単位で移動することもできます。ESC f で一単語分先に進み、ESC b で一単語分前に戻ります。

注意： 日本語については、単語の切れ目を認識することはできませんが、
疑似的な文節を単語の切れ目としています。

>> ESC f や ESC b を何回かタイプしてみなさい。C-f や C-b と併用してみなさい。

C-f や C-b に対する、ESC f や ESC b の類似性に注目しましょう。多くの場合、ESC <文字>は文書関係の処理に使われ、一方 C-<文字>はそれよりももっと基本的な対象（文字とか行とか）に対する操作に使われます。

C-a と C-e も知っていて便利なコマンドです。C-a はカーソルを行の先頭に移動させ、C-e は行の末尾に移動させます。

>> C-a を２回、それから C-e を２回入力してみなさい。これらのコマンドを２回以上繰返しても、カーソルはそれ以上移動しないことに注意。

あと二つ、簡単なカーソル移動コマンドがあります。ファイルの先頭に移動する ESC < と、ファイルの末尾に移動する ESC > です。

テキスト中でカーソルの存在する位置を「ポイント」と呼びます。言いかえれば、カーソルは、テキストのどこにポイントがあるかを画面上で示しているのです。

以下に単純な移動操作について要約します。このなかには、単語や行単位での移動コマンドも含まれています。

C-f　一文字先に進む
C-b　一文字後に戻る

ESC f　一単語先に進む
ESC b　一単語後に戻る

C-n　次の行に移動
C-p　前の行に移動

ESC n　段落の終わりに移動
ESC p　段落の先頭に移動

C-a　行の最初に移動
C-e　行の最後に移動

ESC <　ファイルの最初に移動
ESC >　ファイルの最後に移動

>> 各々のコマンドを試してみなさい。これらのコマンドは、最もしばしば使われるものです。最後の二つでは、この場所とは離れたところに移動するので、 C-v　や ESC v を使ってここに戻って来るようにしなさい。

Emacs の他のコマンドと同様に、これらのコマンドには、繰り返しの回数を指定する引数　を与えることができます。そのためには、コマンドを入力する前に、C-u に続いて繰り返す回数を入力します。

例えば、C-u 8 C-f　とすると、8 文字分先に移動します。

>> C-n あるいは C-p　に適当な引数を指定して、一回の移動でなるべくこの行の近くに来るようにしてみなさい。

このことは C-v　や ESC v についても同様です。この場合、指定された回数分の画面を移動することになります。

>> C-u 3 C-v　と入力してみなさい。

元に戻るには、ESC v を使えばよいのです。

中止コマンド
============

C-g というコマンドで、入力を必要とするようなコマンドを中止することができます。例えば、引数を入力している途中や、２つ以上のキー入力を必要とするコマンドを入力している最中に、それをやめたくなったら、C-g を使えば良いのです。

>> C-u 100 をタイプして引数を１００に設定し、C-g をタイプしなさい。そのあとで C-f をタイプしてみなさい。何文字移動しましたか？もし間違って ESC を入力してしまった時も、C-g を入力すれば取り消せます。

エラー
======

時には、Emacs で許されていない操作をしてしまうことがあります。例えば、コマンドの定義されていないコントロール・キーを入力してしまった時には、Emacs はベルを鳴らし、さらに、画面の一番下に、何が悪かったかを表示します。

Emacs のバージョンによっては、この入門編に書かれていることを実行できない場合があり得ます。その様な場合には、エラーメッセージが表示されますから、何かカーソル移動キーを押して、その次の部分に進んで下さい。

ウィンドウ
==========

Emacs は幾つものウィンドウを持つことと、そのそれぞれに対してテキストを表示することができます。ヘルプや、幾つかのコマンドからの出力を表示するために現れた余分なウィンドウを消すために、次のコマンドを知る必要があります。

C-x 1 ウィンドウを１つにする。

C-x 1 は、他のウィンドウを消して、カーソルのあるウィンドウを、画面全体に広げます。

>> カーソルをこの行に持ってきて、C-u 0 C-l とタイプします。

>> C-h k C-f とタイプしなさい。新しいウィンドウが C-f コマンドのドキュメントを表示するために現れると同時に、このウィンドウがどのように縮むかを観察しなさい。

>> C-x 1 とタイプして、ドキュメントの現われていたウィンドウを消しなさい。

挿入と削除
==========

テキストをタイプしたければ、単にそれをタイプするだけで構いません。目に見える文字（'A','7','*',' あ' など）は Emacs によってテキストであるとみなされ、そのまま挿入されます。行の終わりは改行文字で表され、これを入力するには <Return> をタイプします。

直前に入力した文字を削除するには、<Delete> を入力します。<Delete> は、キーボードで「Delete」と書いてあるキーを押して入力します。「Delete」のかわりに「Rubout」と書いてあるかも知れません。より一般的には、<Delete> は、現在カーソルのある位置の直前の文字を削除します。

>> 文字をいくつかタイプし、それからそれらを <Delete> を使って削除しなさい。

>> 右マージンを越えるまでテキストをタイプしなさい。テキストが一行の幅以上に長くなると、その行は画面からはみ出して「継続」されます。右端にある'\' 記号は、その行が継続されていることを表しています。Emacs は、現在編集中の位置が見えるように行をスクロールします。画面の右あるいは左の端にある'\' 記号は、その方向に行がまだ続いていることを表しています。

これは、文章で説明するより実際にやった方がよく判るでしょう。

>> 先ほど入力した、継続された行の上にカーソルをもっていき、C-d でテキストを削除して、テキストが一行に収まるようにしてみなさい。継続を表す'\' 記号は消えましたね。

>> カーソルを行の先頭に移動し、<Delete> を入力しなさい。これはその行の直前の行句切りを削除するので、その行が前の行とつながってしまいます。つながった行が画面の幅より長くなると、継続の表示がされるでしょう。

>> <Return> を押して、もう一度行句切りを挿入しなさい。

Emacs のほとんどのコマンドは、繰り返しの回数を与えることができます。このことは、文字の挿入についても当てはまります。

>> C-u 8 * と入力してみなさい。どうなりましたか。

二つの行の間に空白行を作りたい場合には、二番目の行の先頭に行き、C-o を入力します。

>> 適当な行の先頭に行き、そこで C-o を入力してみなさい。

これで、Emacs で、テキストを入力し、また間違いを修正するもっとも基本的な方法を学んだことになります。文字と同じ様に、単語や行も削除することができます。削除操作について要約すると次のようになります。

<Delete>	カーソルの直前の文字を削除
C-d	カーソルのある文字を削除
ESC <Delete>	カーソルの直前の単語を削除
ESC d	カーソル位置以降にある単語を削除
C-k	カーソル位置から行末までを削除

何かを削除した後で、それを元に戻したくなることがあります。Emacs は、一文字よりも大きい単位で削除を行った時には、削除した内容を保存しておきます。元に戻すには、C-y を使います。注意したいのは、C-y を削除を行った場所だけではなく、どこにでも出来ることです。C-y は、保存されたテキストを現在カーソルのある場所に挿入するためのコマンドですから、これを使ってテキストの移動を行うことができます。

削除を行うコマンドには、"Delete" コマンドと、"Kill" コマンドとがあります。"Kill" コマンドでは削除されたものは保存されますが、"Delete" コマンドでは保存されません。ただし、繰り返し回数が与えられると、保存されます。

>> C-n を２回ほどタイプして、画面の適当な場所に移動しなさい。そして、C-k で、その行を削除しなさい。

一回目の C-k でその行の内容が削除され、もう一度 C-k を入力すると、その行自身が削除されます。もし、C-k に繰り返し回数を指定した場合には、その回数だけの行が（内容と行自身とが同時に）削除されます。

今削除されたテキストは、保存されているので、それを取り出すことができます。そのためには、C-y をタイプします。

>> C-y を試してみなさい。

C-k を何度も続けて行うと、削除されるテキストは、まとめて保存され、C-y で、その全てが取り出されます。

>> C-k を何度もタイプしてみなさい。

>> テキストを取り出すには、C-y です。カーソルを数行下に移動させ、もう一度 C-y をタイプしてみなさい。これでテキストのコピーができるわけです。

現在何かのテキストが保存されていて、さらに他のテキストを削除するとどうなるでしょうか。C-y は、もっとも最近削除されたものを取り出します。

>> 行を削除し、カーソルを移動させ、別の行を削除しなさい。C-y を行うと、２番目の行が得られます。

取り消し (UNDO)
==========

いつでも、テキストを変更したけれども、それをもとに戻したいときは C-x u で直ります。普通は間違えたコマンドを無効にする働きをします。繰り返して UNDO を行なおうとする時は、何度もそのコマンドを行なえば出来るようになっています。

>> この行を C-k で消して下さい。そして、C-x u で戻して下さい。

C-_は、UNDO を行なう、もう一つのコマンドです。機能は、C-x u と同じです。

C-_や C-x u に UNDO の回数を、与えることが出来ます。

ファイル
========

テキストへの変更を永久的にするためには、それをファイルに保存しなければなりません。保存されないと、ほどこした変更は、Emacs を終了すると同時に失われてしまいます。

いま見ているファイルに対して、あなたの編集を行ったものを書き込みます。いま見ているファイルとは、簡単にいえば編集しているファイル自体のことです。

あなたがファイルをセーブ（保存する）するまで、今までの変更は編集しているファイルに書き込まれる事はありません。それは、あなたがそのように行いたくないのに、途中まで変更を加えたものが勝手に書き込まれるような事がないようにです。

セーブを行った後でさえ変更したものが間違っていた時のために Emacs は名前を変えてオリジナルのファイルを残します。

備考: また、Emacs は不測の事態に対し、一定のタイミングごとに自動的に編集しているファイルの内容を名前を変えたファイルにセーブします。これによって、万一の場合は行ったの変更に対し最小限の被害で済むようになっています。

画面の下の方を見ると、このような感じでモードラインが表示されていると思います。

(例) --**-NEmacs:TUTORIAL (Fundamental) ---55%------[JJJ-]--------

この Emacs チュートリアルのコピーは TUTORIAL と呼ばれています。ファイルをファインド（ファイルを見つけてバッファに読み込むこと）すると、TUTORIAL の部分に表示されます。例えば、new-file という名前のファイルをファインドしたならば、"NEmacs:new-file"というモードラインになるでしょう。

注意： モードラインについては後ほど説明します。少しお待ちを。

ファイルをファインドしたり、セーブしたりするコマンドは、これまでのものとは違い、２つの文字からなっています。C-x に続いて入力する文字が、ファイルに対して行う操作を表します。

もう一つこれまでのものと違う点は、ファインドの時、ファイル名を Emacs に問われます。このことを、端末から引数を読み込んでくるコマンドと言っています。

注意： この場合はファイル名です。

C-x C-f ファイルを見つける（ファインドする）

Emacs はファイル名を聞いてきます。それは、画面の下の行に現れます。ファイル名を指定している部分は、ミニバッファと呼ばれるものです。ミニバッファはこの様な使われ方をします。ファイル名に続いて、リターンキーを押すと、ミニバッファはもう必要ではなくなるので消えてしまいます。

>> C-x C-f とタイプした後に C-g とタイプして下さい。ミニバッファを取り消し、また、C-x C-f コマンドも取り消します。と言う訳で、何もファイルを見つけるようなことはしません。

今度はファイルをセーブしてみましょう。今までの変更を保存するためには次のようなコマンドを使います。

C-x C-s ファイルをセーブする

Emacs の内容はファイルに書き出されます。セーブする一番最初は、オリジナルのファイルは新しい名前をつけられて残されているので内容は失われません。その新しい名前はオリジナルのファイルの名前に'~' をつけたものです。

セーブが終わると、Emacs はセーブしたファイルの名前を表示します。

>> C-x C-s とタイプしてチュートリアルのコピーをセーブして下さい。その時、画面の下の方に"Wrote ...../TUTORIAL"と表示されます。

新しいファイルを作る時、あたかも以前からあったファイルをファインドするようなフリをします。そうして、そのファインドしたファイルにタイプしていきます。

ファイルをセーブしようとした時に、Emacs は今まで編集していた内容を始めてファイルの中に書き込みます。

バッファ
========

もし、２番目のファイルを C-x C-f で取り出すと、１番目のファイルは Emacs 内部に残ります。Emacs 内部にあるファイルからテキストを読み込んで保存しているものはバッファと呼ばれます。ファイルの取り出しは、Emacs 内部に新しいバッファを作ります。

Emacs の中に保存しているバッファのリストを見るには、次のようにタイプします。

C-x C-b　バッファのリスト

>> C-x C-b とタイプしなさい。どのようにそれぞれのバッファは名前を持っているか、そして、どのようなファイル名をつけているのか観察しなさい。

バッファにはファイルと一致ないものもあります。たとえば、"*Buffer List*" というファイルはありません。C-x C-b によって作られたバッファリストに対してのバッファです。

あなたが見ている Emacs ウィンドウ内にある、どんなテキストでも、いずれかのバッファ内にあります。

>> バッファリストを消すため C-x 1 とタイプしなさい。

もし、あるファイルのテキストに変更を行なってから、他のファイルを取り出したとしたら、最初のファイルはセーブされていません。その変更は Emacs 内部のファイルと対応するバッファの中だけに行なわれています。

２番目のファイルに対応するバッファを作ったり、エディットしたりしても、１番目のファイルに対応するバッファには何も影響を与えません。これはとても使い易く、また、１番目のファイルに対応するバッファを取っておくために役に立つ方法です。

C-x C-s でバッファをセーブするために C-x C-f でバッファを切り替えるのは厄介です。そこで

C-x s　現在あるバッファをセーブする。

C-x s は内容を変えたバッファ全てをファイルにセーブします。この時、ひとつひとつの (セーブされるべき) バッファに対して、セーブするか、しないかを y か n で問われます。この表示は画面の下の行に表示されます。例えば、このようにです。

```
Save file  /usr/private/yours/TUTORIAL? (y or n)
```

C-g では Quit してくれません。y か n しか受け付けないので注意してください。

コマンドの拡張

エディタには、コントロール・キーやメタ・キーで入力できるものよりもずっと多くのコマンドがあります。これらを扱うために、拡張（eXtend）されたコマンドを行います。それには、以下の２つの種類があります。

C-x　文字による拡張。続けて一文字を入力します。
ESC x　名前による拡張。続けてコマンドの名前を入力します。

これらは一般に、便利だけれども、これまで見てきたものほど頻繁には用いられないコマンドのためのものです。C-x C-f （ファインド）や C-x C-s（セーブ）はこの仲間です。他に、C-x C-c（エディタの終了）もそうです。

C-x コマンドは、たくさんありますが、便利なのは以下のものでしょう。（ここでは、紹介だけします。）

C-z は Emacs を抜けるのに良く使われる方法です。Emacs を終了することなく、一旦、csh のレベルに戻るには一番良い方法と言えるでしょう。C-z を行なわれても Emacs はストップしているだけで、内容が破壊されるということはありません。

注意: ただし X-window で行なっている場合、もしくは使用しているシェルが sh の時は、この限りではありません。

C-x C-f　ファイルの編集（Find）
C-x C-s　ファイルの保存（Save）
C-x C-c　エディタを終了する。ファイルの保存は、自動的には行われません。しかし、もしファイルが変更されていれば、本当に終了したいのかどうかを聞いてきます。保存して終了する普通の方法は、C-x C-s C-x C-c とすることです。

名前による拡張コマンドには、あまり使われないものや、特定のモードでしか使わないものなどがあります。これらのコマンドは普通ファンクションと呼ばれます。例として、"apropos" をとりあげます。このファンクションはキーワードを入力させ、それにマッチする全てのファンクションの名前を表示します。ESC x とタイプすると、スクリーンの下に ":" が表示されます。これに対して、実行するファンクションの名前（今の場合、"apropos"）を入力します。"apr" まで入力した後スペースを入れれば、後の部分は自動的に補われます。この後、キーワードを聞かれますから、知りたい文字列をタイプします。

>> ESC x をタイプし、続けて、"apropos<Return>" あるいは "apr<Space><Return>" とタイプします。次に、"kanji<Return>" とタイプします。

現れた「ウィンドウ」を消すには、C-x 1 とタイプします。

モードライン

==========

もしゆっくりとコマンドを打ったならば、画面の下底のエコーエリアと呼ばれる場所に打ったものが表示されます。エコーエリアは画面の 1 ばん下の行です。そのすぐ上の行は、モードラインと呼ばれています。モードラインはこんな風に表示されているでしょう。

```
--**-NEmacs:TUTORIAL          (Fundamental) ---NN%----[JJJ-]-------
```

注意: NN%の NN は数字が入っています。あなたが使っている Emacs のモードラインと違うかも知れないけど、慌てないように。例えば、時間や uptime が表示されているのは、display-time という機能が動いているからです。

この行によって多くの有用な情報が得られます。

今、あなたが見ているファイル名を表示しています。NN%は現在スクリーン上にファイルの一番上から何パーセント目が表示されているかを示しています。ファイルの一番最初を表示しているならば、--Top--と表示されてます。ファイルの一番最後を表示しているならば、--Bot--と表示されます。画面の中にファイルの全てが表示されているならば、--All--と表示されます。

モードラインの小括弧の中は、今どんなモードに入っているかを示しています。現在は、デフォルトの Fundamental に入っています。これもメジャーモードの一例です。

Emacs は Lisp mode や Text mode のようなことなるプログラム言語やテキストに対してエディットを行うための幾つかのメジャーモードを持っています。どんな時でも必ずいずれかのメジャーモードの状態になっています。

それぞれのメジャーモードは幾つかのコマンドを全く違う振る舞いにしてしまいます。例を上げてみましょう。プログラムの中にコメントを作るコマンドがあります。コメントをどの様な形式にするかは、各プログラム言語によって違いますが、それぞれのメジャーモードは、きちんと入れてくれます。

それぞれのメジャーモードに入るためのコマンドはモード名の拡張されたものになっています。例えば、M-x fundamental-mode は Fundamental に入るためのものです。

もし、英語をエディットするならば、Text mode に入ります。

>> M-x text-mode <Retuen>とタイプしなさい。

現在のメジャーモードについてのドキュメントを見たい時は、C-h m とタイプします。

>> C-h m を使って Text mode と Fundamental mode の違いを調べなさい。

>> C-x 1 でドキュメントを画面から消しなさい。

一番右端には漢字コード体系に関するフラグの状態が表示されています。NEmacs は、ファイル入出力、入力、画面出力について、それぞれ独立に漢字コードを指定させることが出来ます。

>> モードライン上に"[JJJ-]","[SSS-]","[EEE-]"か"['JSEN-' いずれかの組み合わせ]"が表示されているかどうか確認しなさい。

順にファイル出力、キーボード入力、表示、プロセス入出力の漢字コードを示しており、J は JIS、S はシフト JIS、E は EUC コード、N は無変換、-は未定を意味して

います。どの漢字コードを使用するか、漢字コードを一切使わないかは、その表示によって表されます。現在は表示されているので漢字コードが使用出来る状態になっています。漢字コードを使うかどうかは、それ自体は ON/OFF のトグルになっています。これは C-x C-k t で切り換わります。

次の例は、一度漢字コードを OFF してから、もう一度 ON を行ってみます。

>> C-x C-k t を 2 度行いなさい。

入力モードが JIS コードの設定となっている時、もしあなたの使っている端末にメタ・キーが付いているなら、エスケープ・キーの代わりにそれを使うことが出来ます。その時、メタ・キーの使い方はコントロール・キーと同様に押しながら文字をタイプします。ESC <文字>も M-<文字>も同じ働きをします。今までの説明で ESC <文字>と行なっていたところが、M-<文字>となります。注意しなければならないのは、シフト JIS や EUC コードの時は使用できません。

漢字コードの切り替えは、各々のバッファに対してのみ有効です。それぞれの、漢字コード指定については、C-h a change <Return>で見ることが出来ます。

>> C-h a change <Return>で出てくるドキュメント中の、change-display-code, change-fileio-code, change-input-code, change-process-code の説明を読みなさい。

検索
=====

文字列を、ファイル内で、前方又は後方に、探す事ができます。検索を始めるコマンドは、カーソル位置以降を検索するならば　C-s、カーソル位置以前ならば C-r です。C-s をタイプすると、エコーエリアに "I - search"という文字列がプロンプトとして表示されます。ESC を押すと、終了できます。

>> C-s で検索が始まります。それから、ゆっくりと 1 文字ずつ"cursor"という単語を入力します。1 文字入力するごとに、カーソルは、どんな動きをしますか?

>> もう 1 度 C-s をタイプすると、次の"cursor"を見つけられます。

>> <Delete>を 4 回入力して、カーソルの動きを見なさい。

>> ESC を押して、終了します。

探したい文字列をタイプ中でも、タイプした文字部分だけで、検索を始めます。

次の文字を探すには、再び C-s をタイプします。もし、文字列が存在しなかったら、メッセージが表示されます。C-g でも終了できます。

検索実行中に<Delete>を入力すると、検索文字列の 1 番後ろの文字が消えます。そして、カーソルは、前回の位置に戻ります。たとえば、"Cu"とタイプして、最初の"Cu"の位置にカーソルが動いたとします。ここで<Delete>を入力すると、サーチラインの'u' が消え、カーソルは、'u' をタイプする前に、カーソルがあった'c' の位置に、移動します。

検索実行中に、C-s や C-r 以外のコントロール文字をタイプすると、検索は終了します。

C-s は、現在のカーソル位置以降に出てくる検索文字列を探します。もし、前の方を探したかったら、C-r をタイプすることで、逆方向検索ができます。C-s と C-r は、検索の方向が反対なだけで、全て同じ働きをします。

リカーシブ エディティング レベル

ときどき、(不本意に) リカーシブ エディティング レベルと呼ばれる状態に入ることがあります。メジャーモードの小括弧'()' の回りを中括弧'[]' で囲んだものがモードライン上に表示されます。例えば、(Fundamental) と表示される代わりに [(Fundamental)] のようになります。

注意: ここではリカーシブ エディティング レベル自体については説明しません。

リカーシブ エディティング レベルから抜けでるためには、M-x top-level <Return>とタイプします。

>> 試してみて下さい。スクリーンの底に"Back to top level"と表示されます。

本当は、この試みが行われた時は、すでにトップレベルにいたのです。M-x top-level は、何も影響を与えていません。

リカーシブ エディティング レベルから抜け出るのに対しては C-g は効きません。

ヘルプ
======

Emacs には、たくさんの役に立つ機能があり、ここで、すべてを説明することは、不可能です。しかし、まだ知らない多くの機能を学ぶためには、<HELP>と呼ば

れる C-h をタイプすることで、たくさんの情報を手に入れることができます。

使い方は、C-h をタイプし、続いて必要なオプションを 1 文字タイプします。わからなければ、C-h ？ とタイプすると、どんなオプションがあるのか表示されます。もし、C-h をタイプしてから気が変わったら、C-g をタイプすれば、取り消すことができます。

最も基本的なものは、C-h c です。これに続いてキーを入力すると、そのコマンドについての短い説明を表示します。

>> C-h c C-p とタイプしてみなさい。"C-p runs the command previous-line"のようなメッセージが表示されるはずです。

見たことはあるが、覚えてはいないコマンドも思い出せるのです。C-x C-s のような複数で 1 つのコマンドも C-h c の後ろに続けられます。

もっと詳しく知りたかったら、c の代わりに k を指定します。

>> C-h k C-p とタイプしてみなさい。

Emacs のウィンドウに、コマンドの名前と機能が表示されます。読み終えたら、C-x 1 とタイプすると、抜けられます。

他にも役に立つオプションがあります。

C-h f　　ファンクション名を入力すると、ファンクションを表示します。

>> C-h f previous-line をタイプし、<Return> を押しなさい。C-p コマンドを実行するファンクションについての情報を表示します。

C-h a　　キーワードを入力すると、名前にそのキーワードを含む、全てのコマンドを表示します。これらのコマンドはすべて ESC x で実行できます。

>> C-h a file とタイプし、<Return>を押しなさい。名前に"file"という文字を持つ全てのコマンドを表示します。また、find-file や write-file という名の C-x C-f や C-x C-w のようなコマンドも表示されます。

おわりに
========

忘れずに：　　終了するには、C-x C-c とします。

この入門編は、まったくの初心者にもわかりやすいようにと意図しています。
ですから、もし何かわかりにくい点があったなら、一人で愚痴を言うのではなく、
文句をつけて下さい。

もし、EMACS を何日かでも使ってみれば、それをやめてしまうことなどできなくなるでしょう。最初は戸惑うかも知れません。しかし、それはどんなエディタでも同様です。EMACS のように、非常に多くのことができる場合には特にそうでしょう。そして、EMACS では、実際、何でもできるのですから。

謝辞
=====

この文書は、JUNET で流された"日本語 MicroEMACS (kemacs) 入門編" を GNU-macs (NEmacs) の Tutorial 用に書き換えたものです。

Jonathan Payne による "JOVE Tutorial" (19 January 86) を変更したものであり、それはもともとは、CCA-UNIX の Steve Zimmerman によって変更された、MIT の "Teach-Emacs" 入門編 (31 October 85) を (さらに) 変更したものでした。

Update - February 1986 by Dana Hoggatt.

Update - December 1986 by Kim Leburg.

Update/Translate - July 1987 by SANETO Takanori

特別な謝辞
==========

最初にこれの日本語訳を作られた、SANETO Takanori さん。この文章は GMW + Wnn + NEmacs を使って書きました。そのような素晴らしいプログラムを作った方々へ感謝の意を表したいと思います。翻訳とか、入力とかを色々と手伝ってくれた、藤原祥子さん、どうもありがとう。

誤訳、嘘、その他、の文責は、以下の者にあります。

鈴木裕信 hironobu@sra.co.jp

```
Update/Add - December 1987 by Hironobu Suzuki
Update/Add   November 1989 by Ken'ichi Handa
```

以上が nemacs で esc X help-with-tutorial-for-nemacs と入力すると表示されるファイルである。emacs を使用する上で最初に必要なことが書いてある。

第3章　Mathmatica の組み込み関数

この節の内容は次の２節を読みながら読むと読みやすい。

Mathematica の関数、定数は大文字で始まる。例えば、`Plot[Sin[x],{x,0,2*Pi}]` などをおもいだしてほしい。関数ごとに引数の数が決まっている。上の `Plot` では２個である。引数はコンマで区切る。いまから解説する組み込み関数だけでも、Mathematica を電卓的に使用することができる。さらに、これらの関数を組み合わせて自前の関数を定義することができる。これが、 Mathematica によるプログラミングである。

さらに、Mathematica は、いろいろな省略記法をそなえている。この、省略記法のおかげで簡潔な表現が可能になる。たとえば、 `1+2` は `Plus[1,2]` の省略記法である。

3.1　演算子、定数

3.1.1　簡単な計算をするための関数

`+,-,*,/`　　足し算、引算、かけ算、割り算
実例。

`5*11` は
`55` を戻す。

`^`　　巾乗。
実例。

`2^3` は
`8` を戻す。

`=`　　代入する。
実例。

`x = 2`
は変数 `x` に `2` を代入する。
`hehehe = -5`
は変数 `hehehe` に `-5` を代入する。

`N[ a ]`　　a を小数になおす。桁数を決めたいときは `N[a,n]` とする。　n 桁の小数になおす。
実例。

```
In[45]:= N[Sin[Pi/3]]
Out[45]= 0.866025
In[46]:= N[Pi]
Out[46]= 3.14159
In[47]:= N[Pi,30]
Out[47]= 3.14159265358979323846264338328
```

`a // N`　　a を小数になおす。`N[a]` の後置表現。`//` は前置表現を後置表現になおすためにもちいる。　一般に `f` を関数とするとき `f[a]`　と `a //f` は同じ意味となる。
実例。

```
Sin[Pi/3] //N
```

`p += q, p -= q, p *= q`　　それぞれ $p = p + q, p = p - q, p = p * q$ の意味。
実例。　略

`p++, p--`　　それぞれ $p = p + 1, p = p - 1$ の意味。
実例。　略

`Floor[k]`　　数 k を切り捨てて整数部分のみとりだす。
実例。

`Floor[5/2]`　は
`2` をもどす。

`Mod[k,m]`　　k 割る m のあまり。$0 \leq \texttt{Mod[k,m]} < m$ である。
実例。

`Mod[7,3]` は
`1` を戻す。

`Abs[r]`　　r の絶対値を戻す。
実例。

`Abs[-2/5]` は
`2/5`　を戻す。

`FactorInteger[k]`　　整数 k を素因数分解する。
実例。　略

`Cos[x]`　　x の cos を戻す。
実例。　略

`Sin[x]` x の sin を戻す。
実例。 略

`Sqrt[x]` $\sqrt{x}$ を戻す。
実例。 略

3.1.2 置換をする関数

`ReplaceAll[ a, r ]` または `a /. r` a をルール r で書き換えたものを戻す。ルールは { p1 -> q1 , ..., pn -> qn } の形であたえる。
実例。

```
x^2+y+z /. {x->1,y->3}
```
は、x に 1 を代入し y に 3 を代入して
```
4+z
```
を戻す。

3.1.3 True, False を戻す関数

`p == q` p と q が等しいか？等しければ True を、等しくなければ False を、どちらともいえないときは p == q を戻す。
実例。

```
1 == 2 は
False を戻す。
x == y^2 + 3 は
x == y^2 + 3 を戻す。
```

`p != q` p と q は等しくないか？等しくなければ True をもどす。等しければ False を戻す。
実例。

```
2 != 3 は
True を戻す。
```

`<,>, <=, >=` 数学の $<, >, \leq, \geq$ に対応。
実例。

```
2 >= 5  は
False を戻す。
```

`TrueQ[a]`　　a が True なら True を、そうでないなら False を戻す。
実例。

```
Clear[x]
TrueQ[ x == 1 ]
```
は False を戻す。

`NumberQ[a]`　　a が数なら True を、そうでないなら False を戻す。
実例。 略

3.1.4 論理演算子

`&&`　　かつ (and)
実例。

```
(1 == 3) && (5 != 3)
```
は False を戻す。

`||`　　または (or)
実例。

```
(1 == 3) || (5 != 3)
```
は True を戻す。

`Not[p]`　　p の否定をかえす。
実例。

```
Not[ 1 == 3] は
True
```

3.1.5 数に関してその他

`Clear[x]`　　変数 x に入っている値を忘れる。
実例。

```
In[1]:= x = 99
Out[1] = 99
In[2]:= x-1
Out[2]= 98
In[3]:= 2*x
Out[3]:= 198
```

```
In[4]:= Clear[x]
Out[4]=
In[5]:= x-1
Out[5]= x-1
```

`Pi` 円周率 π
実例。 略

`E` 自然対数の底 e
実例。 略

`I` $\sqrt{-1}$
実例。 略

`SeedRandm[]` 乱数の初期値をかえる。
実例。 略

`Radom[dataType,{ min, max }]` `dataType` (Integer, Real など) の乱数を `min` より `max`] の範囲で作る。
実例。

```
Random[Integer,{0,10}]
```

`Sum[f,{k,min,max}]` $\sum_{k=min}^{max} f$ を計算する。
実例。

```
In[1]:= Sum[k^2,{k,1,10}]
Out[1]= 385
```

`Timing[f]` f を評価してその評価に要した時間とその結果を戻す。
実例。

```
In[4]:= Timing[NIntegrate[E^(-x^2),{x,0,Infinity}]]
Out[4]= {0.916667 Second, 0.886227}
```

例題 3.1 [02] $2^{2^4}+1$ が素数であることを確かめよ。

入力例 3.1

```
In[14]:= FactorInteger[10]
Out[14]= {{2, 1}, {5, 1}}            10 の素因数分解
In[15]:= FactorInteger[8]
Out[15]= {{2, 3}}                     8 の素因数分解
In[16]:= a = 2^(2^4)+1
Out[16]= 65537
In[17]:= FactorInteger[65537]
Out[17]= {{65537, 1}}
```

図 3.1: ListPlot

図 3.2: Point

3.2　グラフを書く関数

`ListPlot[{ y1, ..., yn } ]`　　点 y1, ..., yn をプロットする。
実例。

```
ListPlot[{ {0,0},{1,1},{2,4} } ]
```

これらの点をなめらかにつなぎたいときは

```
ListPlot[{ {0,0},{1,1},{2,4} } , PlotJoined -> True ]
```

とすればよい。

図 3.1 を見よ。

`Show[` *graphic objects* `]`　　*graphic objects* を表示する。*graphic objects* は以下に説明するような graphic primitive を `Graphics[]` の引き数と与えたものである。
実例。

```
Show[ Graphics[{ Point[{0,0}], Line[{{1,0},{0,1}}]}] ]
```

図 3.2 を見よ。

`Point[ {x,y} ]`　　点 (x, y)。
実例。　略

`Line[ { {x1,y1}, ..., {xn,yn} } ]`　　点 $(x1, y1), \ldots, (xn, yn)$ を結ぶ線分。
実例。

```
a = Graphics[{Point[{0.8,0.5}], Line[{{0,0},{1,0}}]}]
Show[a]
```

図 3.3 を見よ。

`Polygon[ { {x1,y1}, ..., {xn,yn} } ]`　　点 $(x1, y1), \ldots, (xn, yn)$ でかこまれた多面体。
実例。　図 3.4 をみよ。

図 3.3: Line

図 3.4: Polygon

図 3.5: Graphics3D

`Graylevel[i]` 塗りつぶしのときのこさの指定。$0 \leq i \leq 1$ であり、 0が黒で1が白。
実例。 略

`PointSize[s]` Point[] でうつ点の大きさの指定。$0 \leq s \leq 1$ であり、 1が画面全体の大きさを表す。
実例。 略

`Thickness[t]` Line[] で描く線分の太さの指定。$0 \leq t \leq 1$ であり、 1が画面全体の大きさを表す。
実例。 略

`Plot[f,{x,xmin,xmax}]` 関数 f のグラフを $x \in [xmin, xmax]$ の範囲で描く。
実例。 略

`Plot3D[f,{x,xmin,xmax},{y,ymin,ymax}]` 関数 f のグラフを $(x, y) \in [xmin, xmax] \times [ymin, ymax]$ の範囲で描く。
実例。 略

`Graphics3D[]` 三次元のグラフィックオブジェクト。
実例。 略

`Line, Point, Polygon` の3次元版
実例。 図 3.5 を見よ。

問題 3.1 [02] $y = |x^2 - 2x + 3|$ のグラフを書きなさい。

問題 3.2 [03] 図 3.6 の図形を描け。

3.3 多項式を扱う関数

`Expand[p]` p を展開する。
実例。 略

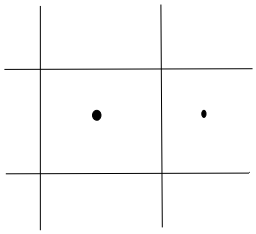


図 3.6: 練習

`Factor[p]`　　p を因数分解する。
実例。 略

`Together[p]`　　p を通分する。
実例。 略

`Numerator[p]`　　分子をとりだす。
実例。 略

`Denominator[p]`　　分母をとりだす。
実例。 略

`Cancel[p]`　　分母分子に共通因子があれば約分する。
実例。 略

`Coefficient[p,t]`　　多項式 p の中の t についての係数をとりだす。
実例。 略

`D[f,x]`　　f を x で微分する。
実例。 略

`D[f,{x,n}]`　　f を x で n かい微分する。
実例。 略

`Integrate[f,x]`　　f を x で積分する。
実例。 略

`NIntegrate[f,{x,xmin,xmax}]`　　f を $x \in [xmin, xmax]$ で数値積分する。つまり

$$\int_{xmin}^{xmax} f(x)dx$$

を計算する。
実例。 略

例題 3.2 [02]

$$x = 3, y = 4, z = 5$$

とおくと

$$x^2 + y^2 = z^2$$

を満たすことを確かめよ。

入力例 3.2

```
In[5]:= x =3
Out[5]= 3
In[6]:= y =4
Out[6]= 4
In[7]:= z=5
Out[7]= 5
In[8]:= x^2+y^2-z^2
Out[8]= 0
```

例題 3.3 [02]

$$x = \frac{2t}{1+t^2}, y = \frac{1-t^2}{1+t^2}$$

とおくと

$$x^2 + y^2 = 1$$

を満たすことを確かめよ。

入力例 3.3

```
In[9]:= Clear[t]
In[10]:= x = 2*t/(1+t^2)
          2 t
Out[10]= ------
                2
         1 + t

In[11]:= y = (1-t^2)/(1+t^2)

                2
         1 - t
Out[11]= ------
                2
         1 + t

In[12]:= x^2+y^2-1

                    2            2 2
                4 t       (1 - t )
```

```
Out[12]= -1 + --------- + ---------
                    2 2         2 2
             (1 + t )    (1 + t )
In[13]:= Together[%]
Out[13]= 0
```

ここで % は直前の結果、つまり Out[12] のこと。

例題 3.4 [05]　$\phi = (x^2 + y^2 + z^2)^{-1/2}$ は点電荷の作るポテンシャルである。$\Delta\phi = 0$ となることが、電磁気の理論でしられているがこれを確かめよ。ここで、

$$\Delta = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$$

である。

入力例 3.4

```
In[19]:= Clear[x]; Clear[y]; Clear[z];

In[20]:= pp = (x^2+y^2+z^2)^(-1/2)

                1
Out[20]= ------------------
                2    2    2
         Sqrt[x  + y  + z ]

In[21]:= rr = D[pp,{x,2}]+D[pp,{y,2}]+D[pp,{z,2}]

                  2                  2                  2
               3 x                3 y                3 z
Out[21]= ----------------- + ----------------- + ----------------- -
           2    2    2 5/2     2    2    2 5/2     2    2    2 5/2
         (x  + y  + z )      (x  + y  + z )      (x  + y  + z )

             3
>    -----------------
       2    2    2 3/2
     (x  + y  + z )

In[22]:= Together[rr]
Out[22]= 0
```

3.4　ベクトルと行列を扱う関数

{a1, ..., an }	配列やベクトルを表す。リストの特別なもの。

実例。

```
a = { 1,5,2 }
```

`a[[k]]` 配列 a の k 番目の元をあらわす。
実例。

```
a = {1,5,2 } のとき
a[[1]] は 1 。 a[[2]] は 5。 a[[3]] は 2。
a[[2]] = 0 とすると
a = {1,0,2} となる。
たとえば次のようになる。
In[24]:= a = {1,5,2}
Out[24]= {1, 5, 2}
In[25]:= a[[1]]
Out[25]= 1
In[26]:= a[[2]]=0
Out[26]= 0
In[27]:= a
Out[27]= {1, 0, 2}
```

`Range[k]` おおきさ k の配列をもどす。
実例。

```
a = Range[3] とすると
a は {0,0,0} となる。
```

`{a1, ..., an} + {b1, ..., bn}` ベクトルとしての足し算をする。
実例。

```
a = {1,5,2}
b = {3,6,2}
a+b は {4,11,1} を戻す。
たとえば
In[28]:= a ={1,5,2}
Out[28]= {1, 5, 2}
In[29]:= b = {3,6,2}
Out[29]= {3, 6, 2}
In[30]:= a+b
Out[30]= {4, 11, 4}
```

`c*{a1, ..., an}` スカラーとベクトルのかけ算をする。
実例。

```
a = {1,5,2}
c = 2
c*a は {2,10,4} を戻す。
```

`{{a11,...,a1n}, ..., { an1, ..., ann}}`　　行列を表す。リストのなかにリストがあることに注意。
実例。 略

`a[[i,j]]`　　行列の (i, j) 成分をとりだす。より正確にいうとリスト a の i 番目の成分をとりだしそのとりだしたリストの j 番目の成分を取り出す。
実例。

```
c = {{1,0,0},
     {0,3,1},
     {0,0,1}}
とおくと
c[[3,2] = -1,
c[[3,3]] = 2 とおくと c は
    {{1,0,0},
     {0,3,1},
     {0,-1,2}}
となる。
```

`MatrixForm[c]`　　行列 c を見やすく表示する。
実例。 略

`c . a`　　行列の積を計算する。
実例。 略

`Tramspose[m]`　　m の転置を戻す。
実例。

```
Transpose[{{1,2},
           {3,4}}]
は  {{1,3},
     {2,4}} 。
```

`Inverse[m]`　　m の逆行列を戻す。
実例。 略

`Det[m]`　　m の行列式を戻す。
実例。

```
 Det[{{1,2},{3,-1}}]
```
は
```
 -7
```
を戻す。

`Eigenvalues[m]` m の固有値を戻す。
実例。

```
In[32]:= Eigenvalues[{{3,4},{2,3}}]

          6 + 4 Sqrt[2]  6 - 4 Sqrt[2]
Out[32]= {-------------, -------------}
                2              2
```

`Eigenvectors[m]` m の固有ベクトルを戻す。
実例。 略

`IdentityMatrix[n]` $n \times n$ の単位行列を戻す。
実例。 略

例題 3.5 [05]

$$[ij] = \begin{vmatrix} x_i & x_j \\ y_i & y_j \end{vmatrix}$$

とおくとき、

$$[23][14] - [13][24] + [12][34] \equiv 0$$

となることを確かめよ。

入力例 3.5

```
In[42]:= v1 = {x1,y1};v2={x2,y2};v3={x3,y3};v4={x4,y4};
In[43]:= Det[{v2,v3}]*Det[{v1,v4}]-Det[{v1,v3}]*Det[{v2,v4}]+Det[{v1,v2}]*Det[{v
3,v4}]
Out[43]= (-(x3 y2) + x2 y3) (-(x4 y1) + x1 y4) -
>    (-(x3 y1) + x1 y3) (-(x4 y2) + x2 y4) +
>    (-(x2 y1) + x1 y2) (-(x4 y3) + x3 y4)

In[44]:= Expand[%]
Out[44]= 0
```

3.5　リストを扱う関数

配列やベクトル、行列はリストの特別なものである。たとえば、{ 1, { 2,3 } } はベクトルでも行列でもないがリストである。

`a[[k]]`　　リスト a の k 番目の要素を戻す。
実例。

```
a = { 1, {2,3}, "cat"}
```
とおくと
`a[[1]]` は 1,
`a[[2]]` は {2,3},
`a[[3]]` は "cat" を戻す。

`First[a]`　　リスト a の 1 番目の要素を戻す。 `a[[1]]` と同じ。
実例。 略

`Rest[a]`　　リスト a より 1 番目の要素をとったものを戻す。
実例。

```
Rest[{cat,dog,3}]
```
は
```
{dog,3}
```
を戻す。
```
Rest[{cat,{c,a,t},3}]
```
は
```
{{c,a,t},3}
```
を戻す。
```
Rest[{cat}]
```
は
```
{}
```
を戻す。

`Append[a,b]`　　a に b をくっつけたリストを戻す。
実例。

```
 Append[{1,2},{3,4}]
```
は
```
 {1,2,3,4}
```
を戻す。
```
 Append[{1,{cat,2}},{{3,dog},4}]
```
は
```
 {1,{cat,2},{3,dog},4}
```

を戻す。
Append[{1,{cat,2}},4]
は後ろがリストでないのでエラー。
Append[{},{3,4}]
は
{3,4}
を戻す。
Append[{1,2},{}]
は
{1,2}
を戻す。

Join[a,b] リスト a と b をくっつけたものをかえす。
実例。

Join[{ t,h,i,s},{i,s}] は
{t,h,i,s,i,s}
を戻す。

Length[a] リスト a の長さを戻す。
実例。

Length[{1,2}]
は
2 を戻す。
Length[{1,{cat,dog},2}]
は
3 を戻す。4 ではないことに注意。

Insert[list,element,n] リスト list の n 番目に element をいれたものを戻す。
実例。

Insert[{this,a,cat},is,2] は
{This, is, a, cat} を戻す。

Drop[list,n] リスト list の n 番目の元をとったものを戻す。
実例。

Drop[{a,b,{c,d},e},2] は
{a,{c,d},e} を戻す。

Complement[u,s] 集合 u から s の元をとりのぞいたものを戻す。
実例。

Complement[{a,b,c},{c,d}] は
{a,b} を戻す。

問題 3.3 [10]
a = {{cat,3},{5},19}, b = {3,{5,dog}}　とするとき、次の関数の値を求めよ。

1. First[a]
2. Rest[a]
3. Rest[First[a]]
4. Join[a,b]
5. Append[a,lion]
6. First[{}]
7. Rest[{}]

3.6　入出力と文字列の処理

Print[f]　　f を表示する。
実例。

Print[hehehe] は
変数 hehehe の内容を表示する。

<< fname　　ファイル fname より入力する。ファイルからの入力をあたかもキーボードから入力されたかのように評価する。
実例。

ファイル sancho.m には
x = 5
y = x*x
と書いてあるとする。
<< sancho.m
を評価すると
変数 x に 5 が、 y に 25 が代入される。

Input[]　　データを入力する。データの型は処理系が適当に解釈する。
実例。

```
hou = Input[]
```
データを読み込み変数 hou へ代入する。

InputString[] データを読み込む。読み込んだデータを文字列 (String) として戻す。
実例。略

WriteString[s] 文字列 s を文字列として出力する。
実例。 略

Characters[s] 文字列 s を文字のリストに直す。
実例。

```
In[33]:= Characters[‘‘Hellow World’’]
Out[33]= {H, e, l, l, o, w,  , W, o, r, l, d}
In[34]:= %[[2]]
Out[34]= e
```

stdout 標準出力をあらわす定数。
実例。 略

3.7 Mathematica の評価の仕組み

Mathematica は

1. 読み込み
2. 評価
3. 評価がかえした値の表示

というサイクルを繰り返している。In[1]:=, In[2]:=, ... ではじまる部分が読み込んだものである。COMMAND + RETURN により読み込んだものの評価をおこなう。評価がかえした値は Out[1]=, Out[2]=, ... ではじまる部分である。どの様な値をかえすかは関数により決められている。たとえば Expand[f] を評価すると式 f を展開した結果を戻す。x = 1 を評価すると 1 を戻す。さらにシステムは x に 1 が入っていることを記憶する。忘れさせるには Clear[x] を評価すればよい。x を評価すると x になにか値が入っていればその値を戻し（上の場合なら 1）、なにも入っていなければ x を戻す。Out[1]=, Out[2]=, ... で表示されるものと、Print[] で表示されるものは別の性格を持っていることを理解しておこう。入力の最後に ; をつけると評価の戻り値を表示しない。例えば、x = 1 を評価すると 1 を戻すが x = 1; を評価しても何ももどってこない。

例題 3.6 [02] ; をつけるのとつけないのの違いを試してみよ。

入力例 3.6

```
In[35]:= Expand[(x+y)^2];
In[36]:= Expand[(x+y)^2]

           2             2
Out[36]= x  + 2 x y + y
```

```
In[37]:= a = Expand[(x+y)^2];
In[37]:= a

          2            2
Out[41]= x  + 2 x y + y
```

１章で簡単にみたように、１から１００までの数の和を求めるのに次の２通りのプログラムの書き方があった。いま説明した Mathematica の評価のやりかたをふまえればこの二つのやりかたの違いがだいたいわかる。（４章、５章を理解すればもっときちんと理解できる。）

```
1.  (あ) re = 0;
    (い) For[i=1,i<=100,i++,
             re = re+i ];
    (う) Print[i];

2.          main[]:=
            Block[{re,i},
    (え)        re = 0;
    (お)        For[i=1,i<=100,i++,
                    re = re+i ];
    (か)        Print[i];
                ]
```

１と２の違いを考えてみよう。１を評価すると（あ）（い）（う）が順番に評価されて、（う）で答えを出力する。２を評価すると Block 以下が main[] という名前の関数として登録される。（え）（お）（か）の評価は行われない。main[] を評価するとはじめて（え）（お）（か）が評価されて（か）で答えを出力する。

3.8　[A] FullForm；すべてはリスト

問題 3.4 [10]　次の関数の値を求めよ。

1. Plus[1,Times[5,2]]
2. First[Plus[x,Times[2,y]]]
3. Head[Plus[x,Times[2,y]]]

問題 3.5 [10]　次の式の FullForm を求めよ。

1. 1+5*2
2. x^2 + y^2 /. { x-> 1}
3. 2/5 //N
4. Factor[x^2 - y^2]

5. `a[[5]]`
6. `x^2 + 1 == 0`
7. `x = 1`

第4章　制御構造

4.1　Flow chat と PAD

4.2　条件判断と繰り返しをする関数

1. For[初期設定, 終了条件, ループ変数の更新, ループ内で実行するコマンド]。引数は４個。
2. If[a,b,c]
3. If[a,b]
4. Which[c1,e1,c2,e2, ..., True,ee]
5. Block[{v1,...,vn},c]
6. Break[]
7. Nest[f,expr,n]
8. Map[f,expr]
9. For[a,b,c,
   e1;
   e2;
   ...
   en;
   ]
10. If[a,b,If[c,d,e]]
11. Block[v,
    e1;
    e2;
    ...
    en;
    ]

4.3　短いプログラム

まずは読み書きそろばんプログラムから。

ファイル cond1.m

```
1: main[]:=
2: Block[{a,b},
3:       a = Input["a="];
4:       b = Input["b="];
5:       Print[a+b];
6:       Print["a kakeru b=",a*b];
7:       ]
```

左のプログラムは二つの数を読み込んでその和と積を出力するプログラムである。3、4 行めで数を読み込む。5、6 行目で和と積を計算して出力する。ひとかたまりのプログラムは Block で囲む。Block の中だけで使う変数は 2 行めのように Block の第一引数のなかで宣言しておく。1 行めはいまのところこう書くんだとおもっておけばよいが、あとで説明するように自前の関数の定義している。実行は自前の関数 main[] を評価すればよい。字さげ (インデントという) をしてわかりやすく書いていることに注意。

実行例

```
In[1]:= <<cond1.m
In[2]:= main[]
a=43
b=900
943
a kakeru b=38700
```

次に If をつかってみよう。

プログラム

```
(* cond2.m *)
main[]:=
Block[{a,b,c},
      a = Input["a="];
      b = Input["b="];
      If[ a> b, c=a, c=b];
      Print[c];
      ]
```

左のプログラムは 2 つの数を読み込んでその最大値をプリントするプログラムである。

出力結果

```
In[1]:= <<cond2.m
In[2]:= main[]
a=4
b=-54354
4
```

cond2.m の改良プログラム

```
main2[]:=
Block[{a,b,c},
      a = Input["a="];
      b = Input["b="];
      c = If[ a> b, a, b];
      Print[c];
      ]
```

If[a,b,c] は a が True のとき b を評価してから値 b を返し、そうでない時は c を評価してから値 c を返す関数である。従って左のように書いた方がより Mathematica らしい書き方である。

次に For を使って繰り返しをやってみよう。

プログラム

```
(* cond3.m *)
main[]:=
Block[{result,k},
      result = 0;
      For[k = 1, k<= 10, k++,
          result = result + k^2;
          ];
      Print[result];
      ]
```

左のプログラムは $\sum_{k=1}^{10} k^2$ を計算するプログラムである。
実は Mathematica にはこういった計算を一気にやる関数もあって、左の In[7] のようにやればよい。

問題 4.1 [05] $\sum_{k=1}^{n} k^2$ の表を $n = 1$ より 100 に対して作れ。

実行例

```
In[5]:= <<cond3.m
In[6]:= main[]
385
In[7]:= Sum[k^2,{k,1,10}]
Out[7]= 385
```

つぎのプログラムは線のひきかたと・の打ち方の解説。

プログラム

```
(* cond4.m *)
main0[]:=
Block[{g},
  g = {};
  g = Append[g,Point[{0.8,0.5}]];
  g = Append[g,Line[{{0,0},{1,0}}]];
  g = Append[g,Line[{{0,0},{1,1}}]];
  g = Append[g,Thickness[0.2]];
  g = Append[g,Line[{{0,0},{0,1}}]];
  Print[g];
  Show[Graphics[g]];
  ]
```

繰り返しをつかってグラフィックスを書く前にちょっとトレーニングを。左のプログラムは・をうって、線分を二本かいてそれから線分の太さを太くしてもう一本線分を書くプログラムである。グラフィックのプリミティブは g のなかに蓄えるようにしている。最後に表示をする。

つぎに、グラフィック画面で For の働きをみてみよう。

```
(* cond5.m *)
main[]:=
Block[{k,g},
  g = {};
  For[k=1,k<=100, k = k+8,
    g=Append[g,Line[{{0,0},{70,k}}]]
    ];
  Show[Graphics[g]];
  ]
```

左のプログラムは傾きが段々おおきくなっていく線分達を描く。
`k = k+8` は `k += 8` と書いた方が簡潔である。

お次はグラフを何枚がつずけて書くプログラムを。

```
(* cond7.m *)
main[]:=
Block[{a},
      For[a = 0.0, a <= 2, a += 0.8,
          Plot[Sin[5*x]+a*Sin[2*x],
               {x,0,10*Pi}]
          ]
      ]
```

このプログラムは

$$\sin 5x + a \sin 2x$$

のグラフを $a = 0, 0.8, 1.6$ について三枚描くプログラムである。

次に `For` の２重ループを作ってみよう

プログラム

```
(* cond6.m *)
main[]:=
Block[{i,j},
      For[i=1,i<=3,i++,
          Print["<<<"];
          For[j=1,j<=2,j++,
              Print["i=",i];
              Print["j=",j];
              ];
          Print["  >>>"];
          ];
      ]
```

`For` の中に `For` を入れることもできる。実行例をよくみて `i`, `j` の値がどう変わっていっているか見て欲しい。

実行例

```
In[21]:= <<cond6.m

In[22]:= main[]
<<<
i=1
j=1
i=1
j=2
  >>>    つずきは右
<<<
i=2
j=1
i=2
j=2
  >>>
<<<
i=3
j=1
i=3
j=2
  >>>
```

次に不定方程式 $x^2 + y^2 = z^2$ の整数解を For を用いたしらみつぶし法で探してみよう。

プログラム

```
(* cond7.m *)
main[]:=
Block[{x,y,z},
      For[x=1,x<10,x++,
          For[y=1,y<10,y++,
              For[z=1,z<10,z++,
                  If[x*x+y*y == z*z,
                     Print[{x,y,z}]
                     ]
                  ]
              ]
          ]
      ]
```

$1 \leq x < 10, 1 \leq y < 10, 1 \leq z < 10$ の範囲で、全部のくみあわせをしらみつぶしに調べてみることにより、整数解を探そうというプログラムである。

問題 4.2 [15]　もっと早く整数解を見つけられるようにプログラムを改良せよ。ちなみに、この方程式の解は理論的によくわかっているので、その結果を使うのは反則です。

実行例

```
In[28]:= <<con7.m
In[29]:= main[]
{3, 4, 5}
{4, 3, 5}
```

配列のなかのデータの最大値を求めてみよう。

```
(* cond8.m *)
main[]:=
Block[{a,k,max,total},
      total:=5;
      a = Range[total];
      For[k=1,k<=total,k++,
          a[[k]]=Random[Integer,{1,100}]
          ];
      Print[a];
      (* search the maximum *)
      max = a[[1]];
      For[k=1,k<=total,k++,
          If[a[[k]] > max,
             max = a[[kk]]
             ];
          ];
      Print["maximum is ",max];
      ]
```

まず、乱数でデータを a へ格納してから、最大値を探す。実行結果は次のようになる。

```
In[36]:= <<cond8.m
In[37]:= main[]
{99, 75, 56, 44, 65}
maximum is 99
```

平均値を求めるプログラムも同じように書ける。

```
(* cond9.m *)
main[]:=
Block[{a,k,sum,total},
      total:=5;
      a = Range[total];
      For[k=1,k<=total,k++,
          a[[k]]=Random[Integer,{1,100}]
          ];
      sum = 0;
      For[k=1,k<=total,k++,
          sum += a[[k]]
          ];
      Print["average is ",N[sum/total]];
      ]
```

左が平均値を求めるプログラムである。実行例は

```
In[38]:= <<cond9.m
In[39]:= main[]
average is 44.2
```

Mathematica らしいプログラムは次のようになる。

```
(* cond10.m *)
main[]:=
Block[{a,k,sum,total},
      total:=5;
      a = Range[total];
      For[k=1,k<=total,k++,
          a[[k]]=Random[Integer,{1,100}]
          ];
      ave=Apply[Plus,a]/total;
      Print["average is ",N[ave]];
      ]
```

問題 4.3 [10]

```
main[]:=
Block[{k},
      For[k=1,k<=5,k++,
```

```
        Print[k]
        ]
    ]
```

1. この関数を While を使って書き換えよ。
2. この関数を Nest, Function を使って書き換えよ。

問題 4.4 [10] `For[k=1,k<=100,k++, x=12; y = x*x]` と `For[k=1,k<=100,k++, x=12345678790123; y = x*x]` の早さを Timing を用いて比べなさい。早さが違うときはその理由も考えなさい。

`If[a,b,c]` の b や c の部分に関数をいくつか書きたいときは; を用いる。たとえば、

```
If[a>b, x=1;
        p=2,
        y = -1;
        q = -2
  ]
```

を評価すると a > b が True の時は x= 1; p = 2 が評価され、そうでないときは y= -1; q = -2 が評価される。

`For[a,b,c,d]` で d の部分にいくつか関数を書きたいときも同様である。たとえば次の例を見てみよう。

```
main[]:=
Block[{re,sq,k},
     re = 0; sq = 0;
     For[k=1,k<=100,k++,
         re = re + k;
         sq = sq + k*k
        ];
     Print[re];
     Print[sq];
     ]
```

この関数は $\sum_{k=1}^{100} k$ および $\sum_{k=1}^{100} k^2$ を計算して答えをプリントする。

`Print["a= "; a; " b= "; b]` とは書かない。これは `Print["a= ",a," b= ",b]` と書くのが正しい。`If[]` の引き数は２個か３個、`For[]` の引き数は４個と決まっているが `Print[]` は可変個の引き数をとる関数である。

問題 4.5 [05] 次のフローチャート（ＰＡＤ）はなにをするか？プログラムになおしなさい。 note 1991/4/27 20:23-

第5章　関数

5.1　自前の関数定義

5.2　短いプログラム

最大値を返す関数を定義しよう。

プログラム

```
(* func1.m *)
max[a_,b_]:=
Block[{},
      If[a>b,
         Return[a],
         Return[b]
         ]
      ]

main[]:=
Block[{p,q},
      p=Input[];
      q=Input[];
      Print[max[p,q]];
      ]
```

実行例

```
In[1]:= <<func1.m
In[2]:= main[]
? 4
? 78
78
```

つぎに、局所変数についての理解を深めよう。

左のプログラムは最大値をかえす関数 max とそれを呼んでいる main とからできている。max はもっと簡潔に、次のように書いても良い。

```
max[a_,b_]:=
If[a>b,a,b]
```

なお Mathematica は次のような組み込みの最大値を返す関数をもっている。

```
In[1]:= Max[{4,78}]
Out[1]= 78
```

プログラム

```
(*func2.m*)
main[]:=
Block[{i},
      i=0;
      Print[i];
      foo[];
      Print[i];
      ]

foo[]:=
Block[{i},
      i=100;
      ]
```

左のプログラムの実行結果は

```
In[48]:= <<func2.m
In[49]:= main[]
0
0
```

となる。けっして、100 とは表示されない。たとえ foo を

```
foo[]:=
Block[{i},
      i=100;
      Return[i];
      ]
```

としたろころで同じである。

次に最大値と最小値を戻す関数をつくろう。考え方は前節のプログラムと同じである。

```
(* func3.m *)
minmax[a_]:=
Block[{max,min,k,n},
      n=Length[a];
      max = min = a[[1]];
      For[k=1,k<=n, k++,
          If[max < a[[k]],
             max = a[[k]]
             ];
          If[min > a[[k]],
             min = a[[k]]
             ];
          ];
      Return[{min,max}]
      ]
```

答えは { 最小値，最大値 } のリストの形にして戻す。このような関数を多値関数と呼ぶ時もある。実行例は以下のとうり。

```
In[2]:= << func3.m
In[3]:= minmax[{1,4,2,6,-3,-2}]
Out[3]= {-3, 6}
```

次に 漸化式 $x_n = 2x_{n-1} + 1,\ x_0 = 0$ できまる数列の n 項めをもとめるプログラムを作ろう。いろいろなやり方でつくってみることにする。

プログラム

```
(* func4.m *)
xn[n_]:=
Block[{k,re},
      re=0;
      For[k=0,k<n,k++,
          re = 2*re + 1
          ];
      Return[re]
      ]

xn2[n_]:=
If[n == 0, 0, 2*xn2[n-1]+1]
```

xn[] はてつずき的にかいたプログラムである。xn2[] は再帰をもちいたプログラムである。実行例は以下のとうり。

```
In[9]:= <<func4.m
In[10]:= xn[1]
Out[10]= 1
In[11]:= xn[10]
Out[11]= 1023
In[12]:= xn2[10]
Out[12]= 1023
```

さて、Mathematica の関数 Nest を用いるともっと簡単にプログラムをかける。さらに、Function の考え方を用いると、以下のようにもっと簡潔にプログラムをかける（次節参照)。

```
In[13]:= f[x_]:=2*x+1
In[14]:= Nest[f,0,10]
Out[14]= 1023
In[15]:= Nest[(2*#+1)&,0,10]
Out[15]= 1023
```

問題 5.1 [10]　アッカーマン関数の値を計算する関数をつくれ。

問題 5.2 [10]

```
foo[n_]:=
Which[0 < n && n < 5, 0,
      n <=0 || 3 < n, 1,
      True,           3
     ]
```

この関数の 1,4,7 における値を求めよ。この関数が 3 を戻すことはあるか？

問題 5.3 [10]　リストの中に、数値データが何個あるかを数える関数 countNumbers　を作れ。たとえば countNumbers[{ 1, cat, 3}] は 2 を戻す。(NumberQ を使う。)

問題 5.4 [10]　リストの中に、リストが何個あるかを数える関数 countLists　を作れ。たとえば countLists[{ {0,1}, cat, {{ 7, 8}, 3} }] は 2 を戻す。(Head を使う。)

問題 5.5 [10]　foo[x_,y_:5]:=x+y　とおくとき f[1,2] および f[1] の値を求めよ。

問題 5.6 [10]　foo[x_Integer]:=x+1;　foo[x_List]:=Append[x,1]　とおくとき f[5] および f[cat,dog] の値を求めよ。

問題 5.7 [10]

```
factorial[x_]:=If[x<=1,1, Times[x,factorial[x-1]]]
```

とおく。Debug[factorial], On[factorial] でトレースをとることにする。factorial[1] および factorial[3] のトレースはどうなるか？

問題 5.8 [10]

問題 5.9 [25]　Well-formed formula とはたとえば、 {or p {and {{not p} q}}} なる形の式だとする。ここで、 p, q は真または偽だとする。常に真である式を恒真式という。与えられた Well-formed formula が恒真式かどうか判定する関数を作れ。

5.3　マルチパラダイム言語 Mathematica

Mathematica をてつずき的にプログラムすることをお勉強してきたがここでは別のスタイルのプログラム方をちょっとかじってみよう。なるべく制御構造や変数を使わず、関数をいっぱい宣言することでプログラムしよう。

問題 5.10 [10]　次の各関数の値を求めよ。

1. `Function[x,Expand[x-4]] [(y+2)^2]`
2. `Apply[ Function[x,x^2],y+2]`
3. `foo[x_]:=x^2 ; Map[foo,{1,2,3,4}]`
4. `Array[Function[x,x^2],5]`
5. `Nest[a=Append[a,Random[Integer,{1,10}]],{},5]`
6. `f[n_]:=1 /; n<=1 ; f[n_]:=f[n-1]*n /; n>1` で `f[0]` および `f[3]`（FullForm で書くと `Condition`）。

問題 5.11 [10]　`Function[#+3]` および `(#+3)&` の FullForm を書きなさい。

5.4　線形サーチ、insert, delete

第6章　行列の処理とリストの処理

6.1　行列

6.2　リスト

問題 6.1 [10]　Append, Length と同じ働きをする関数 myAppend, myLength を作れ。

第7章　入出力と文字列の処理、文字コード、システムプログラムとは？

7.1　１６進法

これは黒板で説明する予定。

7.2　文字コード

文字をどのようにして２進数として表現するかは、規格が決められている。アルファベットについてはアスキーコードがよくつかわれている。アスキーコードでは７ビットを使用してコードを表現する。次の表がその対応表である。

20		40	@	60	‘
21	!	41	A	61	a
22	"	42	B	62	b
23	#	43	C	63	c
24	$	44	D	64	d
25	%	45	E	65	e
26	&	46	F	66	f
27	’	47	G	67	g
28	(	48	H	68	h
29	)	49	I	69	i
2a	*	4a	J	6a	j
2b	+	4b	K	6b	k
2c	,	4c	L	6c	l
2d	-	4d	M	6d	m
2e	.	4e	N	6e	n
2f	/	4f	O	6f	o
30	0	50	P	70	p
31	1	51	Q	71	q
32	2	52	R	72	r
33	3	53	S	73	s
34	4	54	T	74	t
35	5	55	U	75	u
36	6	56	V	76	v
37	7	57	W	77	w
38	8	58	X	78	x

```
39   9      59   Y      79   y
3a   :      5a   Z      7a   z
3b   ;      5b   [      7b   {
3c   <      5c   \      7c   |
3d   =      5d   ]      7d   }
3e   >      5e   ^      7e   ~
3f   ?      5f   _      7f
```

20H が空白、20H 未満は制御コードと呼ばれている。たとえば、0AH が行の区切り、08H が Tab code を表すために使われている。

漢字については、JIS82 コード、シフト JIS コード、EUC コードなどがある。(ねじの大きさと同じでれっきとした JIS 規格がある。本屋さんへいって JIS の規格書 の情報編をみてみよう。）漢字は２バイトつまり１６ビットを使用して表現する。アスキーコードと競合しないためにいろいろな工夫がしてある。ここでは EUC コード表の一部をあげる。EUC コードは MSDOS パーソナルコンピュータ以外のコンピュータでおもに使われているコード系である。

```
b1a1   院
b1a2   陰
b1a3   隠
b1a4   韻
b1a5   吋
b1a6   右
b1a7   宇
b1a8   烏
b1a9   羽
b1aa   迂
b1ab   雨
b1ac   卯
b1ad   鵜
b1ae   窺
b1af   丑
b1b0   碓
b1b1   臼
b1b2   渦
b1b3   嘘
b1b4   唄
b1b5   欝
b1b6   蔚
b1b7   鰻
b1b8   姥
b1b9   厩
b1ba   浦
b1bb   瓜
b1bc   閏
b1bd   噂
```

7.3 入出力関数

OpenRead[], ReadLinst[], Close[], OpenWrite[], WriteString[] など。説明はおさぼり。

7.4 文字列の処理をする関数

StringMatchQ[], ToASCII[], FromASCII[], など。

7.5 ファイルのダンプ

ファイルはディスクの中に格納されている１バイトデータの一次元的な列に名前をつけたものである。プログラム、図形データや文書はファイルとして格納される。

章末のプログラムはファイルの中身をバイトデータの列として表示するプログラムである。こういったことをやるプログラムをダンププログラムという。プログラムの各関数の解説をしよう。

1. toHex[n]：整数 n を１６進法の文字列に直して戻す。
2. toHex2[n]：この関数は toHex と機能は同じ。組み込み関数を利用しているのでとても短い。
3. dump2[]：dum[] の Mathematica らしい書き方。

例題 7.1 [10] 次のようなことが書いてあるファイル t.t のダンプをとってみなさい。

```
Hellow world!
Good by! 12345
```

入力例 7.1

プログラム dump.m は章末にある。

```
In[1]:= <<dump.m
In[2]:= dump2[]
Out[2]= {48   , 65   , 6c   , 6c   , 6f   , 77   , 20   , 77   , 6f   , 72   , 6c   ,
           16     16     16     16     16     16     16     16     16     16     16

>    64   , 21   , a   , 47   , 6f   , 6f   , 64   , 20   , 62   , 79   , 21   , 20   ,
       16     16   16     16     16     16     16     16     16     16     16     16

>    31   , 32   , 33   , 34   , 35   , a   }
       16     16     16     16     16   16

In[3]:= dump[]
Out[3]= {48, 65, 6C, 6C, 6F, 77, 20, 77, 6F, 72, 6C, 64, 21, A, 47, 6F, 6F,
>        64, 20, 62, 79, 21, 20, 31, 32, 33, 34, 35, A}
```

7.6 プログラム dump.m

```
(* dump.m *)
(*  1991/04/08 *)

(* Transform a decimal number into a hexadecimal string *)
toHex[n_Integer]:=
Block[{result,m,r},
      result = {};
      m = n;
      If[m == 0, Return["0"]];
      While[ m != 0,
            r = Mod[m,16];
            result = Append[result, If[r<10,r+ToASCII["0"],r-10+ToASCII["A"]]];
            m = Floor[m/16];
            ];
      result = Reverse[result];
      result = Map[FromASCII,result];
      Return[Apply[StringJoin,result]];
]

dump[]:=
Block[{fp,c,h},
      fp = OpenRead["t.t"];
      c = ReadList[fp,Byte];
      h = Map[toHex,c];
      Close["t.t"];
      Return[h];
      ]

(* example printf["n=``  x=`` \n",1,3] *)
printf[str_,d__]:=
WriteString["stdout",StringForm[str,d]];

(* If we use b^^nnnn and BaseForm[x,b], we can easily do dump without toHex[] *)
toHex2[n_Integer]:=BaseForm[n,16]

dump2[]:=
Block[{fp,c,h},
      fp = OpenRead["t.t"];
      c = ReadList[fp,Byte];
      h = Map[toHex2,c];
      Close["t.t"];
      Return[h];
```

]

7.7 CPU, RAM, BUS, DISK, ethernet, システムコール

第8章　整列：ソート

Mathematica には組み込み関数として Sort がある。ソートをするにはいろいろな方法があり、その計算量も詳しく解析されている。また、ソートのいろいろなアルゴリズムは他の分野のアルゴリズムの設計のよき指針となっている。この章ではソートの仕組みを理解することにしよう。

8.1　バブルソートと クイックソート

章末のプログラムがバブルソートとクイックソートをするプログラムである。

1. データは anArray の中にいれる。サイズは top で指定する。
2. foo1[] で anArray に乱数をいれて初期化する。
3. qSort[p,q] は anArray の p 番めから q 番めまでをクイックソートする。
4. bublsort[p,size] は anArray の p 番めから q 番めまでをバブルソートする。

例題 8.1 [10]　大きさ７のデータと大きさ７０のデータをバブルソート、クイックソートしてその実行時間を調べなさい。

入力例 8.1　まずはデータの数を７として、やってみよう。

```
In[2]:= <<sort.m
In[3]:= top=7
Out[3]= 7
In[4]:= foo1[]
In[5]:= Timing[bublsort[1,7]];
In[6]:= %[[1]]
Out[6]= 0.1 Second
In[7]:= foo1[]
In[8]:= Timing[qSort[1,7]];
In[9]:= %[[1]]
Out[9]= 0.233333 Second
```

というぐあいにバブルのほうが早い。このことから、データが少ない時は単純なアルゴリズムのほうがプログラムが単純になってはやいことがわかる。では、つぎにデータの数を７０としてやってみよう。

```
In[13]:= top=70
Out[13]= 70
In[14]:= foo1[]
In[15]:= Timing[bublsort[1,top]];
```

```
In[16]:= %[[1]]
Out[16]= 19.3667 Second
In[17]:= foo1[]
In[18]:= Timing[qSort[1,top]];
In[19]:= %[[1]]
Out[19]= 3.78333 Second
```

ということで、クイックソートの方が早くなる。

問題 8.1 [10]　top の値をいろいろ変えて計算時間を測定し、グラフに書いてみる。

8.2　計算量（計算料）の解析

my note 1991/4/2 21:12-

8.3　プログラムリスト

```
(* sort.m
   1991/04/08
*)

 top:=7;
anArray = Range[1,top];


foo1[]:=
Block[{k},
      anArray = Range[1,top];
      For[k=1,k<=top,k++,
          anArray[[k]] = Random[Integer,{1,2000}]
          ]
      ]
foo2[] := anArray = {7, 9, 1, 5, 6, 12, 13};

foo3[] := anArray = {1, 7, 7, 4, 17, 3, 16};

(* dqSort[] is for debug. *)
dqSort[p_,q_]:=
Block[{mp,m,b,e,tmp,k},
      If[q-p < 1, Return[]];
      mp = Floor[(p+q)/2];
      m = anArray[[mp]];
      b = p; e = q;
      Print[b,", ",e,",  pivot=",m];
```

```
      For [k=b, k<=e, k++, WriteString["stdout", anArray[[k]],","]];Print[];
      While[True,       (* partition *)
            While[anArray[[b]] < m , b++];
            While[anArray[[e]] > m && b <= e, e--];
            If[b >= e, Break[],
               tmp = anArray[[b]];
               anArray[[b]] = anArray[[e]];
               anArray[[e]] = tmp;
               e--;
               ];
            ]

      If[e < p, e = p];
      For [k=p, k<=q, k++, WriteString["stdout", anArray[[k]],","]];
      Print[":   " ,{p,e},{e+1,q}];
      dqSort[p,e];
      dqSort[e+1,q];
      ]


qSort[p_,q_]:=
Block[{mp,m,b,e,tmp},
      If[q-p < 1, Return[]];
      mp = Floor[(p+q)/2];
      m = anArray[[mp]];
      b = p; e = q;
      While[True,       (* partition *)
            While[anArray[[b]] < m , b++];
            While[anArray[[e]] > m && b <= e, e--];
            If[b >= e, Break[],
               tmp = anArray[[b]];
               anArray[[b]] = anArray[[e]];
               anArray[[e]] = tmp;
               e--;
               ];
            ]

      If[e < p, e = p];
      qSort[p,e];
      qSort[e+1,q];
      ]


bublsort[p_,size_]:=
```

```
Block[{i,j,tmp},
      For[j=size-1,j>p,j--,
          For[i=p,i<j, i++,
              If[anArray[[i]] > anArray[[i+1]],
                 tmp = anArray[[i+1]];
                 anArray[[i+1]] = anArray[[i]];
                 anArray[[i]] = tmp;
                 ];
              ];
          ];
      ]
```

第9章　グラフィックのアルゴリズム

第10章　ヒープ

10.1　ヒープの概念

集合にデータを加えたり (insert[]), 集合のなかにデータがあるかどうか調べたり (find[]), 集合のなかの最小値をとりさる (deletemin[]) などの操作を効率良く行なうためのデータ構造が heap である。これらの操作はいろいろなアルゴリズムにでてくる（たとえば最小木を求めるアルゴリズムをこないだの講義で紹介したが、このアルゴリズムは上の操作の組合せであった)。

1. aHeap　– ヒープを格納する配列。大きさは変数 top で指定する。
2. insert[ee]　– 要素 ee をヒープへ挿入する。現在のプログラムでは ee は正の整数のみ。
3. deletemin[]　– ヒープの一番上（最小値）をヒープから取り去る。
4. find[begin,ee]　– ヒープのなかに要素 ee があるかどうか調べる。あれば True なければ False を戻す。begin から調べ始める。したがって、ふつうは find[1,ee] という形で用いる。
5. foo1[]　– ヒープを初期化する。
6. foo2[]　– ヒープに乱数を用いて作ったデータをいれる。
7. −1　– 空を表すデータ。

例題 10.1 [05]　ヒープを作り、 find[] および deletemin[] を実行してみる。

入力例 10.1

```
In[1]:= <<heap.m
In[2]:= foo1[]
In[3]:= foo2[]
In[4]:= aHeap
Out[4]= {22, 24, 66, 64, 62, -1, -1, -1, -1, -1}
In[5]:= find[1,66]
Out[5]= True
In[6]:= find[1,65]
Out[6]= False
In[7]:= deletemin[]
In[8]:= aHeap
Out[8]= {24, 62, 66, 64, -1, -1, -1, -1, -1, -1}
```

例題 10.2 [05]　insert の時間を heap.m の insert[] と linearinsert.m の linsert[] の間で比較してみよう。

入力例 10.2

```
In[34]:= <<linearinsert.m
In[35]:= top:=10                  anArray の大きさを 10 にする。
In[35]:= makelSample1[]           anArray を初期化する。
In[36]:= anArray                  anArray の中身をみる。
Out[36]= {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1}
In[37]:= makelSample2[]           anArray を 10/2=5 このデータで埋める。
In[38]:= anArray                  anArray の中身をみる。
Out[38]= {8, 17, 53, 65, 75, -1, -1, -1, -1, -1}
In[39]:= linsert[30]              30 を anArray へ挿入する。
In[40]:= anArray
Out[40]= {8, 17, 30, 53, 65, 75, -1, -1, -1, -1}
```

さて使い方になれたところで実行時間の計測をしてみよう。

```
In[61]:= <<heap.m
In[62]:= <<linearinsert.m
In[63]:= top
Out[63]= 10
In[64]:= foo1[]
In[65]:= makelSample1[]
In[66]:= Timing[foo2[]]
Out[66]= {0.2 Second, Null}
In[67]:= Timing[makelSample2[]]
Out[67]= {0.2 Second, Null}

In[68]:= top:=50
In[69]:= foo1[]; makelSample1[];
In[70]:= Timing[foo2[]]            要素数が 50/2=25
Out[70]= {2.31667 Second, Null}    線形 insert
In[71]:= Timing[makelSample2[]]
Out[71]= {6.23333 Second, Null}    heap への insert
In[72]:= Timing[deletemin[]]       heap の deletemin は実行時間にばらつきがある。
Out[72]= {0.283333 Second, Null}
In[73]:= Timing[deletemin[]]
Out[73]= {0.2 Second, Null}
In[74]:= Timing[deletemin[]]
Out[74]= {0.25 Second, Null}
In[75]:= Timing[ldeletemin[]]      線形 deletemin
Out[75]= {0.266667 Second, Null}
```

10.2 プログラム linearinsert.m

```
(* linearinsert.m *)

top:=10;
EMPTY:=-1;
(* anArray *)

makelSample1[]:=
Block[{},
      anArray = Table[EMPTY,{top}];
      ]


makelSample2[]:=
Block[{k},
      SeedRandom[];
      For[k=1,k<=Floor[top/2],k++,
          linsert[ Random[Integer,{1,100}]]
          ];
      ]


lfind[k_]:=
(* linear search *)
Block[{n,i},
      n = Length[anArray];
      For[i=1,i<=n,i++,
          If[TrueQ[anArray[[i]] ==k], Return[k]]];
      Return[Null]]

linsert[ee_]:=
(* linear insert *)
Block[{k,j,n},
      n = top;
      For[k=1,k<=n,k++,
          If[TrueQ[ anArray[[k]] == EMPTY],
             anArray[[k]] = ee;
             Return[];
             ];
          If[TrueQ[ee < anArray[[k]]],
             For[j=n,j>k,j--,
                 anArray[[j]] = anArray[[j-1]]
                 ];
```

```
            anArray[[k]] = ee;
            Return[];
            ];
        ]
    ]
ldeletemin[]:=
Block[{k},
    For[k=1,k<top,k++,
        anArray[[k]] = anArray[[k+1]]
        ];
    anArray[[top]]=EMPTY;
    ]
```

10.3　プログラム heap.m

```
(* heap.m   1991/04/08 *)
(*               04/17 *)

EMPTY := -1;
top := 10;
foo1[]:=
Block[{k},
    aHeap = Range[1,top];
    For[k=1,k<= top,k++, aHeap[[k]] = EMPTY];
    ]

foo2[]:=
Block[{k},
    For[k=1,k<=Floor[top/2],k++,
        insert[Random[Integer,{1,100}]]
        ]
    ]

isEmpty[e_]:=If[TrueQ[e == EMPTY],True,False]



insert[ee_]:=
Block[{k,i,p,tmp},
    For[i=1,i<=top,i++,
        If[isEmpty[aHeap[[i]]],k = i;Break[]]
```

```
        ];
      aHeap[[k]]= ee;
      p = Floor[k/2];  (* p is parent and k is child *)
      While[p>=1 && aHeap[[p]] > aHeap[[k]],
            tmp = aHeap[[p]];
            aHeap[[p]] = aHeap[[k]];
            aHeap[[k]] = tmp;
            p = k;
            p = Floor[k/2];
            ];
      Return[ee];
      ]

deletemin[]:=
Block[{a,b,c,k,i,tmp},
      If[isEmpty[aHeap[[1]]],Return[False]];
      For[i=1,i<=top,i++,
          If[isEmpty[aHeap[[i]]],k = i;Break[]]
          ];
      k--;
      aHeap[[1]] = aHeap[[k]];
      aHeap[[k]] = EMPTY;
      (* Print[aHeap]; *)
      k=1;
      While[k<=top,
            a = aHeap[[k]];
            If[2*k>top, b = EMPTY,b = aHeap[[2*k]]];
            If[2*k+1>top, c = EMPTY,c = aHeap[[2*k+1]]];
            (* Print[a,'' '',b,'' '',c,'' '',k]; *)
            Which[isEmpty[b] && isEmpty[c],Return[],
                  isEmpty[b] && c>a, Return[],
                  isEmpty[b] && c<=a,tmp = c; c=a; aHeap[[k]]=tmp;k=2*k+1,
                  b>a && isEmpty[c],Return[],
                  b<=a && isEmpty[c],tmp=b;b=a;aHeap[[k]]=tmp;k=2*k,
                  b>a && c>a, Return[],
                  b<c, tmp = b; aHeap[[2*k]]=a; aHeap[[k]]=tmp;k=2*k,
                  (* we use EMPTY=-1*)
                  True, tmp = c; aHeap[[2*k+1]]=a; aHeap[[k]]=tmp;k=2*k+1
                  ];
            (* Print[aHeap]; *)
            ];
      ]
```

```
find[begin_,ee_]:=
Block[{},
      If[begin>top, Return[False]];
      If[aHeap[[begin]] > ee, Return[False]];
      If[aHeap[[begin]] < ee,
         If[find[2*begin,ee], Return[True],
            Return[find[2*begin+1,ee]]
            ]
         ];
      Return[True]
      ]
```

第11章　binary tree

11.1　Insert と Find

章末の binary1.m が binary tree を扱うためのプログラム集である。簡単に説明しよう。

1. `makeSample[n]` は乱数を作る関数が作った n 個のデータを binary tree にしてかえす。
2. `member[aa,k]` は数 k が binary tree `aa` の中にあるかどうか調べる。あれば k を戻しなければ `Null` を戻す（したがって画面には何も表示されない）。

例題 11.1　`makeSampe[]` で binary tree を作り、`member[]` でデータを検索してみよう。

入力例 11.1

```
In[1]:= <<binary1.m
In[2]:= aa = makeSample[4]
Out[2]= {{Null, 446, {Null, 461, Null}}, 686, {Null, 985, Null}}
In[3]:= member[aa,446]
Out[3]= 446
In[4]:= member[aa,447]
In[5]:= TreeForm[aa]
Out[5]//TreeForm=
>   List[|                                    , 686, |                    ]
         List[Null, 446, |                   ]       List[Null, 985, Null]
                         List[Null, 461, Null]
```

例題 11.2　`member[]` の実行時間を binary サーチと linear サーチの間で比較してみよう。

入力例 11.2

```
In[1]:= <<binary1.m                          プログラムを読み込む
In[2]:= Timing[aa = makeSample[100];]        100 個のデータをもつ
                                             binary tree  aa を作る。
Out[2]= {8.5 Second, Null}
In[3]:= Timing[member[aa,105]]               105 は aa の中にあるか？
Out[3]= {0.05 Second, 105}
In[4]:= Timing[member[aa,106]]
Out[4]= {0.0833333 Second, Null}
In[5]:= Timing[member[aa,107]]
Out[5]= {0.0666667 Second, Null}
In[6]:= Timing[bb=makelSample[100];]         100 個のデータをもつ
```

```
                                                  配列 bb を作る。bb は配列
Out[6]= {0.45 Second, Null}
In[8]:= Timing[lmember[bb,105]]                   105 は bb の中にあるか？
                                                  linear サーチをする。
Out[8]= {0.283333 Second, Null}
In[9]:= Timing[lmember[bb,106]]
Out[9]= {0.3 Second, Null}
In[10]:= Timing[lmember[bb,107]]
Out[10]= {0.283333 Second, Null}
```

問題 11.1 [05]: いろいろな大きさの binary tree を作り member[] による検索時間を測定しなさい。大きさを n とするとき n を x 軸, 検索時間を y 軸としてグラフをかきなさい。makelSample[] および lmember[] を用いて線形探索についても同じことをやってみなさい。

注意: Mathematica では破壊的なリストの操作ができないので、効率の良い insert プログラムを作ることができない。したがって makeSample[] の実行には非常に時間がかかる。機械語, C , Pascal, Lisp などを用いると効率のよい insert プログラムを書ける。

問題 11.2 [30] Apart[] を用いると効率の良い insert プログラムがかけそうなきがします。やってみて下さい。

参考学習:

```
In[15]:= Part[aa,1,1] = aa
Out[15]= {{Null, 50, {Null, 461, Null}}, 686, {Null, 985, Null}}
In[16]:= aa
Out[16]= {{{{Null, 50, {Null, 461, Null}}, 686, {Null, 985, Null}}, 50,
>     {Null, 461, Null}}, 686, {Null, 985, Null}}
In[19]:= Apply[Part,{aa,1,2}]
Out[19]= 50
```

11.2 計算量 $O(n \log n)$

11.3 Q & A

疑問 11.1 TreeForm[] でリストを表示しようと思ったがと中で次のようにおりかえされてしまった。

```
Out[8]//TreeForm=

>   List[|                                  , 5,
         List[Null, 3, |                  ]
                       List[Null, 4, Null]

>    |                                  ]
     List[|                   , 9, Null]
          List[Null, 7, Null]
```

どうすればよいか？

答え 11.1 たとえば、使用している display が横に 100 文字以上表示できるなら、`ResetMedium[PageWidth->100]` とすればよい。

11.4 プログラム binary1.m

```
(* binary1.m
   1991/03/11
   See my note 91/2/19
*)
(*
   This file contains fundamental functions
   to manupulate binary trees.
*)

NullQ[a_]:=
If[TrueQ[a == Null],True,False]

insert[a_,k_]:=
(* Insert the element  k  to the binary tree  a   *)
Block[{tmp},
      Which[ a == Null,  Return[{Null,k,Null}],
            a[[2]] <= k, If[NullQ[a[[3]]],
                           Return[{a[[1]],a[[2]],{Null,k,Null}}]];
                         tmp = insert[a[[3]],k];
                         Return[{a[[1]],a[[2]],tmp}],
            a[[2]] >= k, If[NullQ[a[[1]]],
                           Return[{{Null,k,Null},a[[2]],a[[3]]}]];
                         tmp = insert[a[[1]],k];
                         Return[{tmp,a[[2]],a[[3]]}],
            True, Return[Null]]]

member[a_,k_]:=
(* a   is a binary tree.
   If  k  is in the binary tree  a, then member[] returns k.
   In another case, member[] returns Null
*)
Which[
      a == Null,  Return[Null],
      a[[2]] ==k, Return[k],
      a[[2]] < k, Return[member[a[[3]],k]],
      a[[2]] > k, Return[member[a[[1]],k]],
```

```
      True, Return[False]]

makeSample[size_]:=
Block[{result,k},
      SeedRandom[];
      result = Null;
      For[k=0,k<size,k++,
          result = insert[result, Random[Integer,{1,2000}]]];
      Return[result]]

(* ここから先は linear サーチをするプログラム *)

lmember[a_,k_]:=
(* linear search *)
Block[{n,i},
      n = Length[a];
      For[i=1,i<=n,i++,
          If[TrueQ[a[[i]] ==k], Return[k]]];
      Return[Null]]

makelSample[size_]:=
Block[{result,k},
      result = Range[size];
      SeedRandom[];
      For[k=1,k<=size,k++,
          result[[k]] = Random[Integer,{1,2000}]];
      Return[result]]


(* The following implementation is done by suzuki@icluna.kobe-u.ac.jp *)
(*  # is in [2] 2.1.13 *)
makeSample2[size_]:=
Block[{},
      SeedRandom[];
      Nest[insert[#,Random[Integer,{1,2000}]]&,Null,size]
      ]
```

第12章　フラクタル

12.1　C 曲線と Koch 曲線

C 曲線は一本の直線を図 12.1 のように 2 本の直線に置き換える操作を何度も繰り返すことにより得ることができる。

点 x の座標を (a,b), 点 y の座標を (c,d) とするときベクトル $z-x$ はベクトル $x-y$ を 90 度回転したものだから、$\begin{pmatrix} d-b \\ a-c \end{pmatrix}$ に等しい。したがって、点 m の座標は

$$\frac{1}{2}\begin{pmatrix} a+d-b \\ b+a-c \end{pmatrix}+\frac{1}{2}\begin{pmatrix} c \\ d \end{pmatrix}=\frac{1}{2}\begin{pmatrix} a+d+c-b \\ b+a+d-c \end{pmatrix}$$

となる。

図 12.2 のプログラムはこの式を用いて C 曲線を描く。図 12.2 のプログラムの points とは

$$\{\text{ 点, 点, } \ldots \text{, 点 }\},\ \text{点} = \{\text{ x, y }\},\ \text{x,y は数}$$

のことである。

例題 12.1　図 12.2 のプログラムを用いて C 曲線を描いてみよう。

入力例 12.1

```
In[24]:= Show[Graphics[Line[cCurve[{{0,0},{1,0}}]]]]
Out[24]= -Graphics-
```

ここで cCurve[0,0,1,0] は

```
                  1    1     1    1
Out[23]= {{0, 0}, {-, -(-)}, {-, -(-)}, {1, 0}}
                  2    2     2    2
```

を戻している。

例題 12.2　もっと深いレベルの C 曲線を描くにはどのように入力すれば良いか。

入力例 12.2

```
In[25]:= tmp = {{0,0},{1,0}}; For[i=1, i<5,i++,tmp = cCurve[tmp]];

In[26]:= Show[Graphics[Line[tmp]]]
```

これは次のように書いた方が Mathematica らしい。

図 12.1: C 曲線

```
(*  fractal.m *)
cCurve[points_]:=
Block[{a,b,c,d,tmp},
      If[Length[points] < 2, Return[{}]];
      a = points[[1]][[1]];
      b = points[[1]][[2]];
      c = points[[2]][[1]];
      d = points[[2]][[2]];
      tmp = {{a,b},{(a+d+c-b)/2,(b+a+d-c)/2},
               {(a+d+c-b)/2,(b+a+d-c)/2}, {c,d}};
      Return[Join[tmp,cCurve[ Rest[Rest[points]] ]]]]
```

図 12.2: C 曲線を書くための関数

```
tmp = Nest[cCurve,{{0,0},{1,0}},4];
Show[Graphics[Line[tmp]]];
```

例題 12.3　例題 12.2 の For のなかの 5 を 8 にすると、

```
General::recursion: Recursion depth of 256 exceeded.
General::recursion: Recursion depth of 256 exceeded.
General::stop: Further output of General::recursion
     will be suppressed during this calculation.
Hold::argct: Hold called with 0 arguments.
Drop::pos: Cannot drop positions 1 through 1 in Hold[].
```

などといったエラーが出る。これを出ないようにするにはどうすればよいか？

入力例 12.3

Mathematica では再帰の深さを 256 までに制限している。上のエラーはこの制限に引っかかって出たもの。この制限値を大きくするため

```
In[3]:= $RecursionLimit = 10000
```

を実行しておけばよい。

問題 12.1 [10] プログラム 12.2 がもっと効率良く動くように改良せよ。

さて、図 12.2 のプログラムは C 曲線の特殊性を活用しているが一般のいわゆる fractal 曲線を描くには図 12.3 のようなプロッタ関数を用いると便利である。

例題 12.4　このプロッタ関数を用いて C 曲線を描くプログラムを書いてみよう。

入力例 12.4

```
cCurve2[l_]:=
      If[N[l] < 1,              (* この N がないと条件式が正しい値を
                                   戻さずうまくいかない *)
         move[l],
         (* else *)
         turnTo[Pi/4];
         cCurve2[l/Sqrt[2]];
         turnTo[-Pi/2];
         cCurve2[l/Sqrt[2]];
         turnTo[Pi/4]];
 (* 6 ではなくもっと大きい数にすればもっと複雑になる。 *)
foo2[]:=
Block[{},
      home[];
      cCurve2[6];
      penUp[];
      ]
```

```
(* plotter functions *)
(* Global variables are aFigure,aPoint,aPath,dir, pendown *)
home[]:=
Block[{},
      aFigure = {};
      aPoint = {0,0};
      aPath = {};
      penDown[];
      dir = 0;
      ]

penDown[] :=Block[{}, aFigure={};  pendown = True;]
penUp[] := Block[{},
                 pendown = False;
                 aPath = Append[aPath,Line[aFigure]];
                 ]


turnTo[theta_]:= dir += theta;

move[l_]:=
Block[{},
      If[pendown,
         aFigure = Join[aFigure,{aPoint,aPoint + l*{Cos[dir], Sin[dir]}}]];
      aPoint = aPoint + l*{Cos[dir], Sin[dir]};
      ]
```

図 12.3: プロッタ関数

実際に曲線を描くには次のように入力すれば良い。

```
In[58]:= foo2[]
In[59]:= Show[Graphics[aPath]]
```

問題 12.2 [10] Koch 曲線を描くプログラムを作れ。

12.2 ハッチンソンの条件とハウスドルフ次元 [M]

問題 12.3 [10M] C 曲線および Koch 曲線のハウスドルフ次元を計算しなさい。

12.3 シェルピンスキーの Gasket

この節ではシェルピンスキーの Gasket を 2 次元および三次元で描いてみよう。mathematica の関数 `polygon[]` をもちいると容易にこれらを描くことができる。

2 次元のシェルピンスキーの gasket を書くプログラム

```
(*************  gasket *****************)
gasket[x_,y_,z_]:=
Block[{a,b,c},
      If[Abs[N[x[[1]]-y[[1]]]] < 1,
         Return[{Polygon[{x,y,z}]}],
         a = gasket[x,(1/2)* ( x+ y ), (1/2)*(x+z)];
         b = gasket[(1/2)*(x+y), y, (1/2)*(y+z)];
         c = gasket[(1/2)*(x+z), (1/2)*(y+z), z];
         Return[Join[a,b,c]]]
      ]

a := {0,0}
b := {5,0}
c := {2.5, 3}
```

実際に絵を書くには

```
Show[Graphics[gasket[a,b,c]]]
```

と入力すれば良い。

3 次元のシェルピンスキーの gasket を書くプログラム

図 12.4: 2 次元シェルピンスキー gasket

```
quad[t_,x_,y_,z_]:=
{Polygon[{t,x,y}],Polygon[{t,y,z}],Polygon[{t,z,x}],Polygon[{x,y,z}]}

gasket3D[t_,x_,y_,z_]:=
Block[{a,b,c,d},
      If[Abs[N[x[[1]]-t[[1]]]] < 1,
         Return[quad[t,x,y,z]],
         a = gasket3D[t,(1/2)* ( t+ x ), (1/2)*(t+y),(1/2)*(t+z)];
         b = gasket3D[(1/2)*(t+x), x, (1/2)*(x+y),(1/2)*(x+z)];
         c = gasket3D[(1/2)*(t+y), (1/2)*(x+y), y,(1/2)*(y+z)];
         d = gasket3D[(1/2)*(t+z), (1/2)*(y+z), z,(1/2)*(t+y)];
         Return[Join[a,b,c,d]]]
      ]

  tt := {1.5,1.5,2}
 aa:={0,0,0}
 bb:={3,0,0}
 cc:={1.5,2,0}
```

---

1. t, x, y, z には 3 次元空間の座標を与える。
2. `quad[t,x,y,z]` は t, x, y, z を頂点とする四面体を描く。

実際に絵を書くには

```
Show[Graphics3D[gasket3D[tt,aa,bb,cc]]]
```

と入力すれば良い。もっと深いレベルの gasket を描くには

```
Show[Graphics3D[gasket3D[5*tt,5*aa,5*bb,5*cc]]]
```

などと入力すれば良い。

問題 12.4 [20]　フラクタルと乱数を利用して山を描いてみよう。

図 12.5: 3 次元シェルピンスキー gasket

第13章　計算幾何

13.1　凸包

13.2　ヴォロノイ図

13.3　三角形分割

第14章　関数近似

14.1　テイラー展開

例題 14.1 [10]　関数 sin のテイラー展開は

$$\sin x = \lim_{N\to\infty} f_N(x), \quad f_N(x) = \sum_{n=0}^{N} \frac{(-1)^n}{(2n+1)!} x^{2n+1}$$

である。$N = 6, 7, \ldots, 15$ のときの $f_N(x)$ のグラフをかいてみよう。

入力例 14.1　略。

14.2　Padé 近似

関数 $(1-w)^{-1/2}$ のテイラー展開は

$$1 + \frac{1}{2}w + \frac{3}{8}w^2 + \frac{5}{16}w^3 + \frac{35}{128}w^4 + \cdots$$

である。いっぽう Padé 近似は

$$\frac{1+(5/2)x}{1+2x}, \quad \frac{1+(9/2)x+(43/8)x^2}{(1+2x)^2}, \ldots$$

である。

14.3　最良近似

第15章　微分方程式の数値解析

15.1　常微分方程式の数値解

15.2　一階偏微分方程式の差分スキーム

次の問題

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 0, \quad t > 0 \tag{15.1}$$

$$u(x, 0) = \varphi(x) \tag{15.2}$$

の解を差分法で作ってみよう。

note 1991/4/11/8:20

15.3　熱伝導方程式の数値解法, CFL 条件

15.4　行列のコレスキー分解

第16章　群

16.1　群表

16.2　generator への分解, あみだ作り

置換群 S_n の元を置換群の生成元 $(12), (23), (34), \ldots, (n-1, n)$ へ分解するプログラムを作ろう。プログラムリストは章末にある。このプログラムを簡単に解説しよう。

16.3　プログラム

```
(* group.m *)
(* generate S_n.  1991/2/12 *)


two2g[l_]:=
 (* two2g[{2,4}] returns { {2,3},{3,4},{2,3} } *)
 (* two2g[{1,1}] goes to infinite loops *)
 Block[{a,b,result,i},
  a = l[[1]]; b = l[[2]];
  If [ a < b,
     Block[{},
           result = {{a,a+1}};
           For [i = a+1, i<b, i++,
                result = Append[result,{i,i+1}] ];
           For [i= b-2, i >= a, i--,
                result = Append[result,{i,i+1}]];
           Return[result]],
     (* else *)
     Return[ two2g[ {b,a} ] ]]]

tog[l_]:=
 (* decompose to generators.
    tog[{{1,2},{3,5,4}}] returns {{1,2},{3,5},{3,4}} -->
                                 {{1,2},{3,4},{4,5},{3,4},{3,4}}
    tog[{{1},{2,3}}] make a error. *)
 Block[{car,a,result,i},
       If [ Length[l] == 0,Return[ {} ]];
```

```
       car = First[l];
       a = First[car];   (* Ex. car == {3,5,4}, a == 3 *)
       result = two2g[{a,car[[2]]}];
       For[i=2, i< Length[car], i++,
           result = Join[result,two2g[{a,car[[i+1]]}]]];
       Return[ Join[result, tog[Rest[l]]]]]


pick[s_,done_]:= (* s = {1,3,5,4,2}, done={1,2,3}, it returns 4 *)
  Block[{tt},
        tt = Complement[s,done];
        If[Length[tt] != 0, tt[[1]], 0]]

decompose[s_]:= (* decompose s into cyclic substitutions *)
                (* {2,3,1,5,4} returns {{1,2,3},{4,5}} *)
 Block[{ans, done, p, a, result},
       ans = {};
       done = {};
       While[ pick[s,done] != 0,
             p = pick[s,done]; (* p is the first element of cycle *)
             result = {p};
             a = p;
             While[ s[[a]] != p,
                   a = s[[a]];
                   result = Append[result,a]];
             If [ Length[result] > 1,
                 ans = Append[ans,result]];
             done = Join[done,result];
             ];
       Return[ans]
       ]

act0[f_,s_]:= (* act0[{2,3},{{1,2},{3,4}}] *)
  Block[{s0,ans,st,a,b,result,i},
     result={};
     For[i=1,i<=Length[f],i++,
        st = s;
        ans = f[[i]];
        While[Length[st] != 0,
              s0 = First[st];
              If[Length[s0] != 2, Print["error in act0"]];
              a = s0[[1]]; b = s0[[2]];
              ans = Which[ans == a, b,
                          ans == b, a,
```

```
                          True,ans];
              st = Rest[st];
              ];
        result=Append[result,ans]
       ];
     Return[result]]


gMult[f_,g_]:=
Block[{k,result},
      result={};
      For[k=1, k<= Length[f], k++,
          result = Append[result, g[[f[[k]]]]];
      Return[ result ]];
```

16.4 Knuth-Bendix アルゴリズム*

第17章　機械語の入門その１

17.1　8086 のアドレッシング

17.2　やさしいプログラム

17.3　68030

1. -S オプションを使うとアセンブラのソースを出力する。例えば cc -S -c as1.c 。

次のプログラム as1.c

```
main() {
 register i;
 i = 1;
}
```

から実行可能形式のファイルを作ると 49152 バイトの大きさになる。このファイルはいろいろな情報を含んでいるが、i = 1 を実行しているのは 3ED 番地あたりである。（なお最初のデータを０番地め、次を１番地めと数えている。）3ED 番地あたりのダンプをみてみよう。

```
    0:  fe ed fa ce  0  0  0  6  0  0  0  1  0  0  0  2
   10:   0  0  0  6  0  0  2 e8  0  0  0  1  0  0  0  1
   20:   0  0  0 7c 5f 5f 54 45 58 54  0  0  0  0  0  0

  3c0:  e7 80 20 6e ff f0 20 70  8  4 20 2e ff f8 e7 80
  3d0:  22 6e ff f0 20 b1  8  0 52 ae ff f8 60 ce 52 ae
  3e0:  ff fc 60 a4 4e 5e 4e 75 4e 56  0  0 70  1 4e 5e
  3f0:  4e 75  0  0 4e 56  0  0 2f  2 24 2e  0  8 66  6
```

C コンパイラは i = 1 を moveq #1,d0 なるアセンブラ言語に翻訳する。これを機械語に直すと、0111 000 0 , 00000001 となる。 データブックの 329p　ここで、0111 は moveq を、000 はレジスター d0 を、あらわす。00000001 はデータ #1 (=1) である。したがって、3ec, 2ed 番地の 70 01 は moveq #1,d0 を意味する。

第18章　構文解析

18.1　逆ポーランド記法

18.2　再帰降下法

18.3　プログラム minicomp.m

```
(*
 * 字句解析部
 *
 * ch = 最後に読み込んだ文字
 * sy = シンボルのタイプ
 * VARIABLE ==> 変数
 * NUMBER ==> 数字
 * その他 ==> 特殊記号
 * val = そのシンボルに対する値
 * NUMBER の時 ==> 実際の数値
 *
 *)

(*
 * 構文解析部
 *
 * expr[]  : 最上位レベルの式
 * exprterm[] : 項
 * exprfactor[] : 因子
 *
 *)
(*
 *      mini compiler for expression.  expression ---> inverse polish notation
*)

isspace[ch_]:=
If[TrueQ[ch == ToASCII[`` ``]],True,False];

error[str_]:= Print[str];

isalpha[ch_]:=
```

```
Which[ch >=ToASCII[``A'']] && ch <=ToASCII[``Z'']], True,
      ch >=ToASCII[``a'']] && ch <=ToASCII[``z'']], True,
      True, False];

isdigit[ch_]:=
Which[ch >=ToASCII[``0'']] && ch <= ToASCII[``9'']], True,
      True, False];

getchar[]:=
If[ptr > Length[str],-1,++ptr;str[[ptr-1]]];


main[]:=
Block[{},
      (* initialize *)
      initglobal[];
      str = InputString[];
      str = Characters[str];
      str = Map[ToASCII,str];
      insymbol[] ;
      expression[] ;
      If [sy != ToASCII[``;'']],
          error[``expression must be terminated by semicolon'']]
      ]

initglobal[]:=
Block[{},
      VARIABLE= 1;
      NUMBER=   2;
      ch = ToASCII[`` ``] ;
      sy =0;
      value=0;
      ptr=1;
      ]


insymbol[]:=
Block[{},
      While[isspace[ch],
            ch = getchar[]
      ];
      Which[
            isalpha[ch], sy = VARIABLE ;
                         value = ch ;          (* variable name *)
```

```
                        ch = getchar[] ,
            isdigit[ch], sy = NUMBER ;
                        value = 0 ;
                        Label[foo];
                        value = value * 10 + (ch - ToASCII[‘‘0’’]) ;
                        ch = getchar[];
                        If[isdigit[ch],Goto[foo]],
            True,sy = ch ;ch = getchar[]
            ];
      ]
expression[]:=
Block[{},
      expr[] ;          (* top level expression *)
      Print[] ;
      ]


expr[]:=
Block[{ope},
      exprterm[] ;
      While[ sy == ToASCII[‘‘+’’] || sy == ToASCII[‘‘-’’],
            ope = If[sy == ToASCII[‘‘+’’],’’+’’,’’-’’];
            insymbol[] ;
            exprterm[] ;
            Print[ope];
            ];
      ]

exprterm[]:=
Block[{ope},
      exprfactor[] ;
      While [sy == ToASCII[‘‘*’’] || sy == ToASCII[‘‘/’’],
             ope = If[sy == ToASCII[‘‘*’’],’’*’’,’’/’’];
             insymbol[] ;
             exprfactor[] ;
             Print[ope];
             ];
      ]

exprfactor[]:=
Block[{},
        Which[sy == NUMBER, Print[value]; insymbol[],
              sy == ToASCII[‘‘(‘‘], insymbol[] ;
                                   expr[] ;
```

```
                                If[ sy != ToASCII[‘‘)’’],
                                   error[‘‘mismatched paren’’]];
                                insymbol[],
            True,error[‘‘invalid factor’’]]
    ]
```

18.4　オートマトン

18.5　LR パーサ

SLR (simple LR) 構文解析法の説明 (Aho, Ullman 177p より)。次の文法（生成規則）に対する SLR パーサを作ろう。

$$
\begin{aligned}
&(1)\ E \rightarrow E + F\\
&(2)\ E \rightarrow F\\
&(3)\ F \rightarrow F * T\\
&(4)\ F \rightarrow T\\
&(5)\ T \rightarrow (E)\\
&(6)\ T \rightarrow id\ \ (identifier)
\end{aligned}
$$

これは

$$
\begin{aligned}
E &\rightarrow E + F \mid F\\
F &\rightarrow F * T \mid T\\
T &\rightarrow (E) \mid id
\end{aligned}
$$

と書いてもよい。

E は expression, F は factor, T は term の略である。id (identifier) は数である。したがって、$2 + 3 * 5$ とか $(2 + 3) * 5$ とかはこの文法をみたしている。なお $+2$ は満たしていないことを注意しておく。

SLR 法では文法より次のような構文解析表をつくって構文解析をする。

状態	id	+	*	(	)	;	E	T	F
0	s5			s4			s1	s2	s3
1		s6				OK			
2		r2	s7		r2	r2			
3		r4	f4		r4	r4			
4	s5			s4			s8	s2	s3
5		r6	r6		r6	r6			
6	s5			s4				s9	s3
7	s5			s4					s10
8		s6			s11				
9		r1	s7		r1	r1			
10		r1	s7		r1	r1			
11		r5	r5		r5	r5			

この構文解析表をもちいてどの様に構文解析をやるか説明しよう。エラーの処理と OK(受理) の処理は省略してある。

driver:=
00: push[0]; (* 状態 0 をスタックへ積んで置く。*)
01: c = ipop[];
02: state = peek[];
03: action = actionTable[[state,c]];
04: **If** action == 移動 (s) **then**
05: push[c];
06: push[nextStateTable[[state,c]]];
07: **else** (* 還元する。 *)
08: rule = nextStateTable[[state,c]];
09: rule 分 pop[] する;
10: ipush[c];
11: ipush[rule の左辺];
12: **endif**
14: **Goto**[1];

1. `stack` – スタック。
2. `push[c]` – スタック (stack) に `c` を積む。
3. `pop[]` – スタックからデータをえる。
4. `peek[]` – スタックのデータを得る、がスタックからとりだしはしない。
5. `inputStack` – 入力データが入っているスタック。
6. `ipush[c]` – `c` を入力スタック (inputStack) へ積む。
7. `nextStateTable` – 構文解析表。移動のときは 1 から 11 を戻す。還元のときは文法規則の番号を戻す。

8. `actionTable` – 構文解析表。移動 (s)、還元 (r)、受理 (OK)、駄目のどれを戻す。

`03:` を終了した時点での各変数の様子を次の入力データ `2+3*5;` に対して見てみよう。

c	state	action	stack, inputStack
2	I_0	s	{ I_0 } , { +,3,*,5,; }
+	I_5	r	{ I_0, id, I_5 }, { 3,*,5,; }
F	I_0	s	{ I_0 }, { +,3,*,4,; }

次にどの様にして文法より構文解析表を作るのか説明しよう。状態の集合、nextStateTable および Following[非終端記号] の三つのデータを計算すれば構文解析表ができる。

まず、状態の集合の計算法を示そう。生成規則の右辺に · を含む生成規則を項 (item) とよぶ。たとえば生成規則

$$E \to E + F$$

からは

$$E \to \cdot E + F,\ E \to E \cdot + F$$

$$E \to E + \cdot F,\ E \to E + F \cdot$$

の四つの項ができる。直感的には生成規則のどこまでをすでに読み込んだのかを項は示している。

I を項とする。 `Closure[I]` とは I を含む項の集合であり、次の性質を満たすものとする。

$$A \to \alpha \cdot B\beta \in \texttt{Closere[I]}$$

かつ $B \to \gamma$ なる生成規則があれば、

$$B \to \cdot\gamma \in \texttt{Closure[I]}$$

である。

たとえば $E' \to \cdot E$ の Closure を計算すると

$$\begin{aligned} E' &\to \cdot E \\ E &\to \cdot E + F \\ E &\to \cdot F \\ F &\to \cdot F * T \\ F &\to \cdot T \\ T &\to \cdot(E) \\ T &\to \cdot id \end{aligned}$$

となる。これが上の構文解析表の状態 0 (I_0) に対応する。次に id が読まれたとすると状態は

$$T \to id \cdot$$

となるはずである。· の後ろになにもないのでこの状態の Closure はこのままである。この状態の集合は 5 (I_5) に対応する。この状態では `Follow[F]` に属する終端記号が次に待っていれば $F \to id$ の還元をおこなう。以下、省略。

状態 I_k が項

$$A \to \alpha \cdot \beta\gamma$$

を含むとする。`nextStateTable[[` I_k`,` β`]]` は

$$Closure[A \to \alpha\beta \cdot \gamma]$$

である。

例として、`nextStateTable[[` I_0`, ( ]]` を計算してみよう。

$$T \to \cdot(E) \ \in I_0$$

だから

$$T \to (\cdot E)$$

の Closure が求めるものである。Closure は

$$\begin{aligned} T &\to (\cdot E) \\ E &\to \cdot E + F \\ E &\to \cdot F \\ F &\to \cdot F * T \\ F &\to \cdot T \\ T &\to \cdot(E) \\ T &\to \cdot id \end{aligned}$$

である。この状態を I_4 となずける。

状態 I_k が

$$A \to \alpha\cdot$$

を含むとする。次の入力記号が `Follow[A]` ならルール

$$A \to \alpha$$

で還元 (r) する。

状態 I_k が

$$E' \to E\cdot$$

を含むとする。次の入力記号が ; (終わりマーク) ならば受理 (OK)。

問題 18.1 [15]　次の生成規則に対する構文解析表を作れ。

$$\begin{aligned} E &\to E + F \mid F \\ F &\to (E) \mid id \end{aligned}$$

第19章　エラー

19.1　NeXT のエラー

疑問 19.1　NeXT の反応がとってもおそくなったり、グラフをかいていたら突然止まってしまったり、メモリーがないから書けないよ、といってきたりする。どうしてか？

答え 19.1　原因は “貧乏” である。すなわち、RAM とハードディスクの不足でこのようなことがおこる。したがってこれらの部品を買い足せばよい。または、機械がすいているときに使えば良い。

19.2　Mathematica のエラー

疑問 19.2　1+5 のような簡単な式の評価をしようとしても Running ....　といったきり返事をしない。なぜか？

答え 19.2　Mathematica の Kernel が動いていない可能性がある。原因は一意的に決められないが、たとえば Preferences の Start up をみると Kernel が rsh sencha mathremote jasmin などとなっていることがある。このときは default へ戻す。
メモリーが足りない場合もある。[2] の 2.8.2 Memory を参照。

さて、Mathematica の言語処理系からのエラーメッセージをみよう。

疑問 19.3

```
In[20]:= Factor[x^2-I*y^2]
                     2       2
Factor::notratpoly: x  + -I y  is not a polynomial with rational coefficients.
                   2       2
Out[20]= Factor[x  + -I y ]
```

答え 19.3　有理数を係数とする多項式しか因数分解できない。$x^2 - iy^2$ は複素数を係数としている。

疑問 19.4

```
In[21]:= foo[n_]:=For[i=1,i<10,i++,n = n+i]
In[22]:= foo[3]
Set::raw: Cannot assign to raw object 3.
Set::raw: Cannot assign to raw object 3.
Set::raw: Cannot assign to raw object 3.
General::stop: Further output of Set::raw
     will be suppressed during this calculation.
```

答え 19.4　関数定義の引数 n には代入できない。実は foo はマクロなので、foo[3] は

```
For[i=1,i<10,i++,3 = 3+i]
```

と同じこと。

疑問 19.5

```
In[23]:= for[i=1,i<10,i++,Print[i]]
2
Out[23]= for[1, True, 1, Null]
```

あれっ。なにもおきないぞ？

答え 19.5　これは、For を for と書き間違えたため。こういったエラーは見つけにくい。

疑問 19.6

```
In[24]:= Block[x=3; Print[x]]
Block::argct1: Block called with 1 argument.
```

答え 19.6　Block の引数は２つである。Block[,x=3;Print[x]] と書くべき。

疑問 19.7

```
In[25]:= foo[]:=Block[{},For[i=1,i<=3,i++,Print[i]]]
In[26]:= For[i=1,i<10,i++,foo[]]
```

このプログラムはいつまでたっても止まらない。

答え 19.7

```
In[25]:= foo[]:=Block[{i},For[i=1,i<=3,i++,Print[i]]]
In[26]:= For[i=1,i<10,i++,foo[]]
```

というぐあいに、i を局所変数とすればよい。

疑問 19.8

```
In[27]:= foo[]:=Block[{k},k=1;k=k*3,Return[k]]
In[28]:= foo[]
Block::argct: Block called with 3 arguments.
```

答え 19.8

```
In[27]:= foo[]:=Block[{k},k=1;k=k*3;Return[k]]
```

が正しい。, と ; の間違い。

疑問 19.9

```
In[29]:= cc={{1,0,0},{0,2,0},{0,0,3}}  とおこう。
Out[29]= {{1, 0, 0}, {0, 2, 0}, {0, 0, 3}}
In[30]:= cc[[1]][[1]]    とすると（1,1）成分をとりだせるが、
Out[30]= 1
In[31]:= cc[[1]][[1]]=9  とは、代入できない。
Part::nonsymb: cc[[1]] in assignment of part is not a symbol.
Out[31]= 9
In[32]:= cc[[1,1]]=9
Out[32]= 9
In[33]:= cc
Out[33]= {{9, 0, 0}, {0, 2, 0}, {0, 0, 3}}
```

答え 19.9

```
In[32]:= cc[[1,1]]=9
Out[32]= 9
In[33]:= cc
Out[33]= {{9, 0, 0}, {0, 2, 0}, {0, 0, 3}}
```

とやればできる。

疑問 19.10

```
In[5]:= <<f2.m          一回目はエラーを起こさない。
In[6]:= <<f2.m          しかし、２回目は、
Set::write: Symbol Real is write protected.
```

となるのはどうしてか？ここで、f2.m には

```
w2 = 2
```

という文が含まれている。

答え 19.10 ２回目では w2 にはすでに 2 がはいっており、$2 = 2$ という代入がおきたため。

疑問 19.11 次のエラーはどうして起こったか？

```
Join::normal: Normal expression expected at position 1 in
    Join[aFigre, {{0, 0}, {5, 0}}].
```

答え 19.11 変数 aFigre にリストがはいっていないから。

疑問 19.12 次のエラーはエラーメッセージからは全く原因が想像できない。なにが原因か？

```
In[21]:= move[4]

Set::write: Symbol Times is write protected.
```

ここで

```
1: move[l_]:=
2:  Block[{},
3:        If[pendown,
4:           aFigure = Join[aFigure,{aPoint,aPoint + l*{Cos[dir], Sin[dir]}}]]
5:        aPoint = aPoint + l*{Cos[dir], Sin[dir]};
6:        ]
```

答え 19.12 これは単に４行目の最後にセミコロンを忘れただけである。

第20章　NeXT, Mathematica 雑学, その他

20.1　間違いやすいところ

```
For[k=1,k<10,k++,Break]
k
```
とすると、
```
2
```
を戻す。

`map[]` と `scan[]`

20.2　snap shot のとり方

20.3　∞

2 , 487p - 545p の解説をする。

1. フラクタル上のラプラシアン。音、グラフ。
2. 数値積分： Euler-Darboux, 2 変数 Legendre の数値計算。
3. yacc : The system Kan.
4. 計算幾何: 毎週金曜午後。ことしの special program.
5. process 間通信、ウインドウプログラム: The system Kan.
6. Math remote プロトコルを調べる。
7. Gnu Mathematica.
8. Lint for Mathematica.