UNIX/Linux 初級講座・起動とコマンド編

廣明　秀一
hiroakih@tsurumi.yokohama-cu.ac.jp
横浜市立大学大学院総合理学研究科生体超分子システム科学専攻
生体超分子計測科学研究室　助教授

濱田　季之
thamada@tsurumi.yokohama-cu.ac.jp
横浜市立大学大学院総合理学研究科生体超分子システム科学専攻
タンパク質フォールド科学研究室　客員研究員
thamada@gsc.riken.go.jp
理化学研究所ゲノム科学総合研究センター
タンパク質構造機能研究グループ
タンパク質機能研究チーム　研究員

---

§1　はじめに・・・UNIX と Linux

科学技術分野ワークステーションおよび通信関連のサーバのオペレーティング・システムとして UNIX（ユニックス）が広く採用されている。UNIX にはさまざまなシステムが含まれており、開発経緯や供給元で微妙に異なる。最近流行の各種メディアにも取り上げられる「Linux」 や 「FreeBSD」も PC に移植された free の UNIX である。それぞれの UNIX で利用方法は少しずつ異なるが、基本的な構成は同じである。計算科学実習室のコンピュータには Linux が install されている。一方、各研究室のワークステーションや測定装置制御用コンピュータには『商用 UNIX』が install されているものが多い。
Linux はフィンランド Helsinki 大学の Linus Torvalds (現在 Transmeta 社社員) によって、DOS/V 互換 PC で稼動する free の UNIX として作成された. その後 Linux のソースコードが公開され、多くのプログラマ（ボランティア）の助力により, Internet を通して発展した.

情報生物学実習の一環として Linux のコンピュータの基本的な使用方法を学ぶ。また、構造生物学実習の一部で NMR の解析にも当実習室の Linux 環境を用いるので、このテキストの内容レベルのことはある程度理解しておいて欲しい。：

1. Linux とは汎用のマルチユーザの会話型の OS(オペレーティングシステム)である。
   UNIX/Linux は複数のユーザが同時に一つのコンピュータにアクセスして、そこで独立して作業できる環境を実現している。(WindowsXP などでは最近はできるようになりつつあるが、実際には PC ではできない)

2. 基本的には CUI（コマンドライン・ユーザー・インタフェース）ベースの OS である。
   どういうことかというと、画面上のアイコンをマウスでクリックしたりするのではなく、コマンドを入力してリターンキーを押すことで操作を行うことになっている。
   もちろんよく使う機能の一部はアイコンやマウスからメニューで利用できるようになっているものも多いがコマンドを覚えて損はないし、コマンドを覚えておかないと使えないことが多い。

3. ネットワークを通じて遠隔地からでもコンピュータの操作ができる。（telnet や ssh によるログイン）

4. ファイルのディレクトリ（directory・folder のこと）はツリー状の階層構造になっている。従ってたくさんのファイルも簡単に仕分けして整理できる。

5. 入出力機器（キーボード，画面，プリンターなど）もファイルとして扱うので入出力の切り替え（redirection，リダイレクション）が簡単にできる。
   【リダイレクションの例】
   cat file1 file2 というコマンド（計算機に対する命令）は file1 と file2 を垂直に連結してその結果を画面に表示するものだ。出力をディスクファイルとして保存したいときは，リダイレクションの記号「＞」を使って

   cat file1 file2 > file3

とすると，画面に表示される代わりに file3 に保存される。プログラム開発の際に画面出力のルーチンだけを書いておけば，必要に応じてリダイレクト機能をつかって output をプリンタまたはディスクに振り分けられる。
同様に入力も「<」の記号を使ってリダイレクトできる。

6. いくつかのコマンドを連続して実行させるとき，UNIX/Linux はコマンド間のデータの受け渡しを見えないファイルを経由して行う（パイプライン：記号「|」）。例えば，
   【パイプの例】
   sort mydatafile > sortfile .... mydatafile の内容をソートして，sortfile に出力する。
   pr sortfile > prfile .... sortfile を整形して prfile に出力する。
   less prfile .... prfile を画面に出力する。
   ↓
   sort mydatafile | pr | less （うえの３行がこうなる）

   と中間ファイルをいちいちセーブしないで、一行の処理として書ける。
   （なお、この例のように入力データを加工して出力するコマンド群をフィルターコマンドという。パイプラインとは，２段，３段重ねのフィルターを設けることに相当する。）

7. コマンドの逐次処理・同時並行処理ができる。
   【いくつかのコマンドを逐次順番に実行】→「；」（セミコロン）で切って並べる

   date; who

   【いくつかのコマンドを並列で実行】→バックグラウンド処理を表す「&」（アンパサンド）記号で区切る

   cc myprog & emacs myprog2.c

   この例では myprog を（裏の処理として）Ｃコンパイラにかけておいて，その間に myprog2.c の編集が（表の処理として）エディタ emacs を使ってできる。

8. コマンド言語が一種のプログラム言語になっている。UNIX/Linux はコマンドをシェル（shell）と呼ばれるコマンドアナライザ（インタプリタ）で解釈する。
   コマンドは実行可能型プログラムのファイル名であってもよいし，一連のコマンドを入れたコマンドファイル名であってもかまわない。
   UNIX のシェルは入力から受け取った文字列について、基本的には二つのことしかしない。
   （１）それがもしシェルの解釈できる制御コマンドまたは記号だった場合には、そのとおり解釈して処理する（たとえば；とか｜とか>とかいくつかのコマンド）
   （２）そうでない文字列が来た場合は、それを実行ファイルの名前と解釈して、いくつかのディレクトリの中にそのファイルがあるかどうか調べて、あったら実行する。
   すなわち，Perl のスクリプトや tcsh のスクリプト（バッチファイル）やコンパイル済のＣ言語などのプログラムをファイルとして（しかるべき場所に）格納しておき，これを実行可能型にしておけば，コマンドとして使用できる。例えば，

```
sort $1 > $1.tmp
sort $2 > $2.tmp
join -j1 1 -j2 2 -o 2.1 1.2 1.3 2.4 2.5 $1.tmp
$2.tmp > joint.data
rm *.tmp
```

   という内容のファイルを joint という名前で登録しておき，

   chmod u+x joint
   で実行可能ファイル形式にしておくと，joint file1 file2 と入力するだけで joint の内容が一括して実行できる（バッチ処理）.

9. いろいろなプログラム言語（C, Fortran77, Pascal, Lisp, Perl, Tcl/Tk）やソフトウェアツール（make など）がそろっている。特に Linux の場合は基本的に無料（インターネットからダウンロードで無償配布されている）で入手でき、その環境で C/C++/Fortran77 のコンパイラが利用できる。最近は Intel や AMD などの PC 用の CPU が安価でしかも高速になってきた。そのため高度な科学技術計算も、ごく一部の専門機関のものではなくなり、家庭用 PC ですら十分研究目的の実用に耐える計算速度が出る。

10. 科学技術計算用ソフトウェアは大半が C/Fortran などで開発されておりソースコードが公開されていたり、学術機関には無償でライセンスが提供されることが多い。したがって研究に必要な計算を行うソフトウェアを入手しコンパイルすることで利用可能となる。逆に、自分でオリジナルなソフトウェアを開発し、それを公開することで世界中の研究者に提供するこ

とができる。
Linux 上で利用可能な科学技術計算ソフトウェアについては
http://sal.linet.gr.jp/sal1.shtml
にデータベースがあるので参考にして欲しい。
またバイオ関係で汎用されるソフトウェアに関して、
http://www.biolinux.org/
にもアプリケーションのまとめやインストールの仕方のガイドがある。

§２　Linux の起動、停止、ログインとログアウト

UNIX/Linux は複数のユーザが使う共同利用棟のようなものだ。だから，自分の領分は自分で管理する必要がある。UNIX/Linux 棟の自分の部屋に入るには自分の鍵を使って錠前を開け閉めしなければならない。錠を開けることをログイン，錠を閉めることをログアウトと言う。

1. 起動
   PC の電源を入れる。
   3 つの利用可能な OS（windows XP, Red Hat Linux 8.0, Turbolinux 10）が縦に表示される。↑↓の矢印キーで、 Red Hat Linux 8.0 を選択し、enter する。起動がはじまる。

2. ログイン（login）
   (# 遠隔にある UNIX/Linux コンピュータに他の端末から接続する場合は、端末プログラム telnet やその端末エミュレーターをスタートさせ目的の装置に接続する。（例　:　telnet pc150.ec.tsurumi.yokohama-cu.ac.jp [return]））

a) 画面に login: と表示されるから自分のログイン ID を入力する。このとき，大／小文字に注意。taro, Taro, TARO は厳密に別ものと解釈される。ログイン ID を入力したら，[return]を押すこと。

b) 画面に Password: と表示されたら自分のパスワードを入力する。このパスワードは画面には表示されない（覗かれないため）。

c) しばらくして，UNIX/Linux はウェルカム・メッセージを表示して，[name@pc150 ~]$ のプロンプト（個人の環境により異なる）が出たら，作業が開始できる。

3. X-window システムの起動 (startx)
   現在の白黒表示の状態のことを「コンソール・モード」と呼ぶ。この状態でも、必要なコマンドは全て入力でき、計算などもできるので、プログラム作成やコンパイルなどをこのまま行ってもいい。しかし「マルチウィンドウ」の状態にして、複数のターミナル「窓」やソフトウェアを同時に起動して、作業効率をあげることができる。またグラフィックスが利用できるようにするために、UNIX システムに共通のシステム X-Window system というのがあるので、それを起動する。

   [name@pc150]$ startx [return]
   （または startx -- -bpp 16 [return] ←16 ビット色で X を起動するとき。）

   これで、いくつかのアイコンの表示されたカラフルな画面表示に変わったはずである。
   なお、ここでの表示は Linux や UNIX の設定によって変わりうるし、個人での設定も可能である。横浜市大鶴見の実習室 PC の標準的な設定は WindowMaker というウィンドウマネージャが行っている。Linux では Gnome や KDE といったウィンドウマネージャも利用可能であり、それぞれ便利な機能がある。自分で設定して変更できる場合は変更して利用しても構わないが、実習自体は WindowMaker で行うので自己責任で変更して欲しい。

4. X-window システムの停止

   マウスをバックグラウンドの部分に移動　→　右マウスをクリック　→　メニューの一番下「Exit」→「Exit」
   Exit windows manager?と聞かれるので、「Exit」

5. コンピューターの停止または再起動

   コンソールモードにて、キーボードから [CTRL][ALT][Delete] を押す.

   最初の OS 設定画面になったら、 PC の電源を切ることが可能である。再起動の際は，そのまま続ける。

6. ログアウト（logout）
   コンソールモードで作業終了後、別のユーザーに渡す場合はログアウトする。またリモートアクセスの場合はログアウトを行う。多くのワークステーションでは、ログアウトをするものの装置本体は停止しないことが多い。これは複数のユーザーが利用していたり、装置がサーバーとして通信や他の機器を監視していたりするからである。また SGI ワークステーションなどで電源投入時・停止時にシステム破壊が起きることが多かったからである。
   PC の場合には停止でも logout でもどちらでも構わない。

[name@pc150]$ logout[return]
   または

[name@pc150]$ exit [return]

   いずれも小文字で入力することに注意。
   さらに，リモートアクセスの場合は端末プログラムを終了させる。

7. 作業の強制終了（kill）
   何らかの原因で作業が中断してしまう（hang up）してしまうことがある。このようなときは，ちゃんと後始末しなければ他のユーザに迷惑がかかることにもなる。例として，ファイルを表示する命令を中途半端に裏で処理させる状態を作ってみる。
   [name@pc150]$ cat &
   まず作業（process）の状態を見てみよう：
   [name@pc150]$ ps

PIP	TTY	TIME	COMMAND
1653	ttys	10:00	cat
1641	ttys	10:00	ps
1619	ttys	10:00	telnetd
1620	ttys	10:00	tcsh

   などの表示がでるだろう。この例では cat コマンドには 1653 のプロセス番号（PID）で実行したままの状態になっている。このプロセスを停止するためには,

[name@pc150]$ kill 1653   または
[name@pc150]$ kill –9 1653

とする。PID はシステムがその都度実行しているプロセスに割り振る番号である。再び ps コマンドを入力してみると cat コマンドは消滅したので表示されないはずである。べつの方法として、バックグラウンドで動いているプロセスを確認する方法としては

[name@pc150]$ jobs
[1] + Suspended (tty input)
[name@pc150]$ kill %1   ←（%記号に注目）
[1] Terminated

§3　UNIX/Linux 基本コマンド

まず，基本的ないくつかの UNIX/Linux コマンドを説明する。ここでは，コマンドのオプションなどは必要最低限のものしか説明しないので，くわしくは何かの参考書，例えば
井田昌之・田中啓介：UNIX 詳説　－基礎編－，改訂２版，丸善，1990。
村井　純ほか：プロフェッショナル UNIX。アスキー，1986
等を参考にするとよい。研究室のスタッフ・先輩諸氏から使いやすいものを教えてもらうと良い。英語をいとわなければオンライン・マニュアルを見るのがいちばんよい。

これだけは覚えよう！よく使う UNIX/Linux 基本コマンド、早見表

1. man	manual	オンラインマニュアルを表示する
2. pwd	print working directory	現在の作業場所(カレントディレクトリ)を表示する
3. mkdir	make directory	新しいディレクトリを作成する
4. ls	list	現在の作業場所にあるファイルのリストを表示する
5. cd	current/change directory	現在の作業場所(カレントディレクトリ)を移動する
6. cat	concatenate	ファイルの内容を見る(標準出力に流す)
7. more / less		ファイルの内容を｢画面に｣表示する
8. mv	move/rename	ファイルを移動する、あるいはファイル名を変更する
9. cp	copy	ファイルのコピーを作成する
10. rm	remove	ファイルを削除する
11. lpr		印刷する

1. オン・ライン・マニュアルを見る（man）
manua l の略
UNIX/Linux ではマニュアルはシステムにファイルとして入っている。
[name@pc150]$ man man[return]
で，オン・ライン・マニュアル自身の使い方が表示される。あるコマンド，例えば，sort の使い方を知るには，
[name@pc150]$ man sort[return]
とする。man はマニュアル（manual）の略。日本語化の進んでいる Linux では man あるいは jman のコマンドで日本語マニュアルが表示されるので、便利である。Unix/Linux のコマンドには大抵の場合たくさんのオプションがついており、その文法を覚えるのは大変である。そういうのは無理して覚えずに必要なときだけ man コマンドでチェックすればよい。

2. ここはどこ？（pwd）
現在の作業場所（カレントディレクトリ）が分からなくなり，迷子になってしまったときは，print working directory の省略である
[name@pc150]$ pwd
とタイプすると 自分の居場所がフルネームで表示される。実習室の設定では tcsh でのプロンプトがカレントディレクトリを表示している。

わたしは誰？（whoami）
自分のログイン名を忘れてしまったときなどに打ち込む。

3. ディレクトリをつくる（mkdir）
make directory

ファイルを分類して整理しておくために，適宜自分でディレクトリを作成して保存する。例えば，Ｃプログラム・ファイルとデータ・ファイルを別のディレクトリに仕分けしておくなど。
UNIX/Linux は階層ディレクトリ構造をとっている。
実習室の Linux に login すると，/home/s01/name（name は各人のログインＩＤ)，というディレクトリに【居る】ことになる。この場をホーム・ディレクトリという。これは，ルート・ディレクトリ（／），その下に home ディレクトリ，その下に s01 ディレクトリ、その下に name ディレクトリが階層をなして存在することを意味する。
なお，／を一番初めに書くとルートディレクトリの意味になるが，二番目以下はディレクトリ名の区切りを意味する。
プログラムファイルを入れるためのディレクトリ名を prog，データファイル用のディレクトリ名を data として，ディレクトリをつくってみる（make directory）。
[name@pc150]$ mkdir prog
[name@pc150]$ mkdir data
これで２つのディレクトリができた。

【相対パスと絶対パス（フルパス）】
さて、いま自分はホームディレクトリ（/home/s01/name ）にいるので、先ほど作成した prog を list させたければ単に ls prog と打つだけでもよい。しかし、実際には prog は /home/s01/name/prog となる。いいかえると『ファイル名を指定するときに/～自分が今いるディレクトリ名～/までは省略してよい』
/ではじまって、そのファイルに行き着くまでのディレクトリ名をすべて記述する書き方のこと

を絶対 path（絶対パス、フルパス）などとよぶ。
自分がいる場所を基準にファイル名を記述する場合は相対パスという。相対パスにはほかに「../」「./」という書き方がある。
../ → 現在いるところの１個上のディレクトリ
./ → 現在のディレクトリ（特に混乱を避けるために明示的に示すとき）
~/ または ~name/ → ユーザー「name」のホームディレクトリ

4. ディレクトリの内容をみる (ls) list
ls
ls -l
ls -a
mkdir の例を実行したあとで，
[name@pc150]$ l s -l[return]をタイプすると，
drwx---r-x 1 name s01 May 7 14:03:24 prog
drwx---r-x 1 name s01 May 7 14:03:30 data
のように，ディレクトリ /home/s01/name の内容が表示される。この表示の意味は，ディレクトリ（ｄ），所有者に対する属性（読みｒ，書きｗ，実行ｘを許可），同一グループのメンバに対する属性，他者に対する属性（読みｒと実行ｘは許可し，書き込みｘは禁止），別の名前の数（１は別名無し），所有者名，サイズ（文字数），作成の日時，ディレクトリ名（頭にｄの代わりに一が付いているときはディレクトリでなくファイル名），となっている。
引数の-l は long の意味で，このように詳しい情報が出力される。この-l がないと，ファイル名だけが表示される。(-l オプション無しも試してみよう).
[name@pc150]$ ls -l ＜ディレクトリ名＞で任意のディレクトリの内容が表示される。ディレクトリ名を省略した時はカレントディレクトリが仮定されている。
[name@pc150]$ ls -al でドットファイル(隠しファイル)も表示できる。 a は all の意味。
なお，ls コマンドには引数がある．引数はディレクトリの名前である．ディレクトリの名前を指定して ls をすると，自分が作業しているディレクトリ以外のファイルの存在を確認できる．これをうまく使うと目的のファイルがどこにあるか探すことができる．
[name@pc150]$ ls -l /home/s02/*/1.txt
このコマンドは，ファイル名は覚えているが(1.txt)それをどこのディレクトリに仕舞ったか忘れたときに探すことができる．この*はワイルドカードであって全ての文字列に一致するからである．

5. 作業場所を移動する（cd） change directory
サブディレクトリ data を新たな作業場所にして，その下にデータファイルをつくるには，
[name@pc150]$ cd data をタイプしてディレクトリを移動し（change directory），同時に先のリストコマンドを実行する。こんどは，あなたの居場所は/home/s01/name から data になった。フルネームは/home/s01/name/data になる。この作業場所をカレント・ディレクトリ（current directory ）という。このように階層ディレクトリで上からみてサブディレクトリを表すときはカレントディレクトリ名を省略することができる。（ただし，この例のようにディレクトリを下に下に行くときにのみ省略できる）。
ホームディレクトリへは単に
[name@pc150]$ cd とタイプするだけで戻れる。
直上のディレクトリ（この例では /home/s01/name ）は .. （ピリオド２つ）と書くことができる。次の例はサブディレクトリ data から prog ディレクトリに一挙に移る例だ
[name@pc150]$ cd ../prog

ディレクトリをいったりきたり（pushd と popd）
たとえば現在/home/s01/hiroaki/data/nmr/nmrPipe/demodata という長い名前のディレクトリで作業していて、少しだけ/home/share/bin に作業しにいってから、また戻って来たいときは
[name@pc150]$ pushd /home/share/bin
として、なんらかの作業をしてから
[name@pc150]$ popd
としてやると、もとのところに戻ってこれる。
これは shell が一時的に directory 名をメモリに pushd が実行されるたびに貯めこんでいき、popd が呼ばれるたびに前に貯めこんだ directory 名を逆順で取り出してくるのである。

6. ファイルの内容を見る（１） cat
この目的のために UNIX/Linux はいくつかのコマンドを用意している。本来の目的は複数のファイルを垂直に連結する（concatenate）ことだが,
[name@pc150]$ cat mydatafile
でファイル mydatafile の内容を画面に表示する。
複数のファイルを指定するとそれらを順に連結して出力する。
[name@pc150]$ cat file1 file2 file3
この使い方は３つのファイルを３人で手分けし

て作成し，それをまとめるときに有効だ。
リダイレクションの機能をつかうと，cp コマンドと同じことができる。
[name@pc150]$ cat file1 > file2
cat を使用するとき，ファイルの長さが画面に入り切らないほど大きいと，ファイルの前の部分は順送りされて画面から消えてしまう。このようなときは，次に述べる less または more コマンドを使う。

7. ファイルの内容を見る（2） less / more
長いファイル一画面分表示して，そこで中断して指示を待つ。中断しているときはファイル名を反転表示しているので，通常はスペースバーを押す。すると次の一画面を表示して再び止まる。前の画面を見たいときは， b（backward）を入力すると画面は逆スクロールする。less を中止したいときは， q をタイプする。表示が中断しているときに emacs のようなサブコマンドが使用できる。スラッシュ（／）に正規表現（例えば，/abc）で正規表現文字列（abc）を持つ行に画面が移る。次の同一文字列を検索するには emacs のように n をタイプする。より詳しい使い方は，表示が中断しているときに h をタイプすると，ヘルプが出る。
[name@pc150]$ less mydatafile
[name@pc150]$ ls -al mydir | less
以前は more はバック・スクロール機能は持っていなかったが、Linux の more などでは最近は b で戻れるようになった。なお終了するときは q と打ち込む.

8. ファイル名を変更する
ファイルを移動する（mv）
ディレクトリから別のディレクトリにファイルを移動する（move），同じディレクトリ内でファイルを移動する（ファイル名の変更）ときは,
[name@pc150]$ mv ファイル名 ディレクトリ名
[name@pc150]$ mv ファイル名 ディレクトリ名/.
[name@pc150]$ mv 元ファイル名 先ファイル名
とタイプする。移動先に同じファイルがあったときは重ね書きされ，以前のファイルは消失するので注意を要する。あるディレクトリの下に移動することを明示的に示すときは/.を必ずくっつけること.
[name@pc150]$ mv oldfile newfile .... oldfile を newfile に名前を変更
[name@pc150]$ mv mydatafile ../prog/. ..... mydatafile を/home/s01/name/prog に移動
移動後のファイルのフルパス名は
/home/s01/name/prog/mydatafile
になっている
[name@pc150]$ mv mydatafile ../prog/myprog.c .. mydatafile を /home/s01/name/prog に myprog.c という名前に変更して移動する。..はいま自分がいるところの一つ上の階層のディレクトリを示す.

9. ファイルを複写する（cp）
他のユーザのファイルを自分なりに編集するときや，プログラムの改訂をするときは，まずコピー（copy）をつくりる。使い方は mv コマンドとほとんど同じだ（mv では元ファイルは消去されるが，cp では元ファイルは保存される）。
[name@pc150]$ cp oldfile newfile
コピー先に同じファイルがあったときは重ね書きされ，以前のファイルは消失する。そうならないようにするには，set noclobber（「UNIX の環境設定」を参照）と設定してくとよい。なお実習室の設定では
alias cp cp –i　となっているので、同名ファイルがあった場合には書き換えてよいかどうか確認を求めてくる。

10. ファイルを消す（rm）
間違ったファイルを作ってしまって，それを消したいとき（remove），
[name@pc150]$ rm ＜ファイル名＞
とタイプする。このとき複数のファイル名をスペースで区切って入力することができる。実は実習室の設定では rm を rm -i という様に，別名登録（alias）してあって，
[name@pc150]$ rm mydatafile.* とタイプすると，確認を求めてくる，
mydatafile.1 :
ここで本当に消して良いのなら y，良くない（誤ったファイル名を指定してしまってとき）は n をタイプする。すると，次のファイルを確認してくるので同様に答える。
なおディレクトリを消すときには-r option をつける。
rm –r [directory name]

11. 印刷する lpr
lpr –P(プリンタ名)　ファイル名
lpr　ファイル名
でテキストファイルまたはポストスクリプトファイルをプリンタに直送する。

その他の便利なコマンド群

12. ファイルの種類を確認 (file)
UNIX/Linux では ASCII ファイル（人間が意味あるものとして読める内容のファイル）と binary ファイル（計算機にとっては意味があるが，人間にとっては意味あるものとして読めない内容のファイル）を区別しないので，ファイルによっては，画面にでたらめな文字がでるものもある。試みに，file コマンドを使ってみてみよう。UNIX/Linux ファイルのタイプを教えてくれる。
[name@pc150]$ file　ファイル名

13. head と tail
ファイルの頭だけ・終わりだけを表示
これは cat とほとんど同じだが，ファイルの始めまたは終わりの数行を任意に表示させることができる。
次の例は mydatafile の始めの 10 行を表示する。
[name@pc150]$ head -10 mydatafile
次の例は mydatafile の終わりの 10 行を表示する。
[name@pc150]$ tail -10 mydatafile
次の例はパイプラインをつかって，mydatafile の中間の 6～10 行を表示する。
[name@pc150]$ head -10 mydatafile | tail –5

14. シンボリックリンク　(ln -s)
file を link する
ln -s <大元のファイル>　<リンク>
[name@pc150]$ ln -s /home/share/bin/molmol mymolmol
Mac のエイリアス、Win のショートカットに相当する。あるファイルの分身を別のディレクトリに作りたいときに、このようにする。この例では、molmol という実行ファイルのリンクを、現在のディレクトリに mymolmol という名前で作成したので、
[name@pc150]$　./mymolmol
と入力すると molmol が起動する。
Linux では本家(商用)UNIX に存在して Linux には free では存在しないコマンドが多数ある。そのような場合、「ほぼ同じように動作する代用品コマンド」に、本家 UNIX と同じ名前でシンボリックリンクを登録すると、本家 UNIX と同じコマンドの操作性が確保できるため、よく多用される。ためしに、我々が使っているシステムコマンドが格納されている領域　/bin　の中身の sh に関するファイルだけを見てほしい
[name@pc150]$ ls -l /bin | grep sh　←（ファイル名に sh を含むとこだけ抽出！）
Linux で csh と思って使っているものが実は tcsh であり、sh だと思ってつかっているものが実は bash であるということがわかる。
ln -s はウィンドウズコマンドの「ショートカットを作成する」に似ている。

15. 引数のエコー（echo）
コマンド echo の引数，シェル変数，環境変数を標準出力する。

[name@pc150]$ echo $TERM .... 端末名をエコー
[name@pc150]$ echo whoami .... 引数（who am i）を表示
[name@pc150]$ echo \`whoami\` .... コマンド who am i の実行結果を表示（逆コーテーションマーク「 ` 」に注意）

[name@pc150]$ echo 1 2 3 > mydatafile ... 引数（1 2 3 ）をファイル mydatafile に出力
このコマンドはシェルスクリプトのプログラミングのとき，エラーメッセージを表示するようなときに多用される。

16. ファイルの属性の設定，変更 (chmod)
change mode
一般のユーザはシェル・スクリプト・ファイルを作成して，実行許可を与えるときにこのコマンドを使用する。
[name@pc150]$ chmod u+x mysh
シェル・スクリプト mysh の実行許可を自分（u すなわち所有者，作成者）に与える。他のひとにも実行許可権を与えるときは u の代わりに a を使って（ a は省略可能）：
[name@pc150]$ chmod a+x mysh
[name@pc150]$ chmod +x mysh
とする。
[name@pc150]$ chmod 755 mysh などのように、ファイル属性を明示的に決定するやり方もある。

17. パスワードを変更する
(passwd　/ yppasswd)
システム管理者から与えられた仮のパスワードや他人にバレたパスワードは変更しなければならない。実習室の PC は詳しくは述べないが NIS というしくみによってユーザーが管理されているので **yppasswd** を使う。通常のワークステーションでは passwd を使う。

[name@pc150]$　yppasswd
Changing NIS account information for hiroakih on linux....
Old password:　（現在使用中のパスワードを入力する（画面には表示されない）

New password:　（変更したいパスワードを入力してリターンキーを打つ）
Retype new password:　（もういちど新パスワード）

§4　tcsh / csh を上手に使う

UNIX/Linux の特徴である，入出力の切り替え，パイプライン，バッチ処理などは，コマンド解析プログラムともいうべきシェル **shell** がおこなっている。UNIX/Linux システムのシェルには何種類もある。

- sh .... ＡＴ＆Ｔ版 UNIX（System V），Ｂシェル（＄マーク）
- bash .... Linux でデフォルトの sh の上位互換シェル
- csh .... バークレイ版 UNIX，Ｃシェル（%マーク）
- tcsh ....csh に emacs 風の機能を追加したシェル　Linux で csh の代わりとして汎用
- ksh .... sh に emacs 風の機能と csh のコマンド履歴機能などを追加したシェル

sh は UNIX/Linux システムには必ずあるが，tcsh, ksh が存在しない UNIX/Linux システムもある。
sh はファイルも小さく従って実行速度も他のシェルよりも速い特徴がある。最近はコンピュータの性能向上により sh の実行速度は事実上問題にならなくなってきた。csh, tcsh は重いシェルだが、そのぶん機能は豊富で操作性がよく、シェル・スクリプトも読み書きしやすい。なかでも tcsh は，history 機能，コマンド行編集機能に特徴があるので、慣れてくると面倒なキーボード入力を簡略化することができる。したがって初心者にも使いやすくなっている。（以下，コマンドを入力して実行させるには return キーを押すが，[return]の表記は省略する。）

§１の(８)で述べたバッチ処理ファイルをもう一度考えてみる。
sort $1 > $1.tmp
sort $2 > $2.tmp
join -j1 1 -j2 2 -o 2.1 1.2 1.3 2.4 2.5 $1.tmp $2.tmp > joint.data
rm *.tmp
上の内容をエディタでつくり，joint という名前をつけた。これを[chmod +x joint]コマンドで実行可能形式にモードを変更した。これで joint はあたらしいコマンドになった。ところで，chmod をせずにシェルを使ってこのバッチファイルを実行するには，
[name@pc150]$　tcsh joint
[name@pc150]$　tcsh < joint　（入力リダイレクション。joint というファイルの中身をキーボードから打つのと同じ効果）
とすればよい。これは，tcsh に joint というファイルの内容（命令群，プログラム）を実行させることを意味している。シェルをうまく活用すると，C などの言語プログラムでいちいちプログラミングをしなくとも，かなりの仕事をこなすことができる。例えば，tcsh でのシェルスクリプトは次のようなかたちをしている。

```
foreach file ($argv)
gawk '{print"....",$1,$2,...,$13}' $file | pr -h $file
end
```

また，このようにいちいち命令群をファイルとして作成しなくともよい。コマンド待ちのプロンプト（[name@pc150]$　など）が表示されている状態で直接これらのプログラムの入力が可能で
[name@pc150]$ foreach i (1 2 3 4 5)　　　(←＃i という変数に順繰りに 1~5 まで代入しながら以下の操作をする）
? cat file　$i
? end
このシェルプログラムは file1, file2, file3, file4, file5 を表示させるもので，もっと簡単には：
[name@pc150]$ cat file[12345]
とあらわすことができる。これ以上のシェルプログラミングの詳細はここでは省略する。

tcsh のいくつかの便利な機能を説明する：

1. **コマンドの履歴機能（hitory）**
   シェルのプロンプトマーク％に数字がついていると思うが，これはログインしてから入力したコマンド数を表している。history コマンドをタイプして history 機能を使ってみよう。

   ```
   [name@pc150 ~]$ history
   1 17:18  cd data
   2 17:19  ls –o
   3 19:20 pwd
   4 19:21 emacs mydata
   5 19:22 history
   [name@pc150 ~]$ _
   ```

   いままでに入力したコマンドが表示される。この history 機能は以下にのべる事項と密接に関係している。

2. **コマンド行の編集**
   直前のコマンドを修正して再実行
   長いコマンドのたった一文字をミスタイプした

だけでもコンピュータは正しく動いてくれない。tcsh での修正は簡単にできる。

[name@pc150 ~]$ ls /home/share/pdd
ls: /home/share/pdd: No such file or directory（←エラーと表示された（実は pdd でなく pdb））
[name@pc150 ~]$ 「↑」（上矢印記号）を使って，

% 8>↑
と入力すると

[name@pc150 ~]$ ls -l　/home/share/pdd

と直前のコマンドがよみがえる（_はカーソルを表す）。ここで back キーで１文字消して，b をタイプし，return キーを入力する。「↑」を２回押すと２つ前のコマンド（つまり，２つ前の history）がよみがえる。別の箇所を修正したければ，「←」を押してカーソルを戻し，delete キーで編集すれば良い。このように直前のコマンドの修正・再実行に，emacs 風のカーソル・コントロール・キーが利く。

3. いくつか前のコマンドの再実行
これはコマンド歴（history）を「↑」または「↓」でたどる事を繰り返せば良いわけだ。しかし，面倒なら「！」を使う。
[name@pc150 ~]$　!!　直前のコマンドの再実行
[name@pc150 ~]$　!n　n 番目のコマンドの再実行（history コマンドを実行して n を求めれば良い）
[name@pc150 ~]$　!c　c で始まる最新のコマンドの再実行
[name@pc150 ~]$　!n | cat　　n 番目のコマンドの後ろに |cat を付加して再実行
ここで，n は現在のコマンド番号から遡って 20 以内（個人の環境によって異なる）にあるコマンド番号が参照可能だ。

4. 別名機能（alias）
コマンド名に別の簡略化した名前を付けることができる。UNIX/Linux のコマンド名は分かりにくいほど簡略化したものばかりだ。しかしこれを自分専用のコマンド名に変えることができる。例えば，リストコマンドを MS-DOS 風に DIR とすることもできる。

[name@pc150 ~]$　alias dir 'ls -alF'
また，シェルスクリプトを作る代わりに alias 機能を使える。

[name@pc150 ~]$　alias cd 'cd \!* | pwd | ls -o | p'
自分の環境でどの様な別名が登録されているか見てみよう。

[name@pc150 ~]$　alias
l　　ls -lF !* | less
rm　　rm –i
by　　logout
エディタでファイル~/.cshrc（各ユーザのホームディレクトリは~または~name[name=自分の名前]で表すことができる）を編集しておくと，次回以降のログイン時に有効となる。§４の UNIX の環境設定でも alias に触れる。

5. シェル変数
シェルもプログラミング言語の一種なので変数が使える。一般的な使い方は，長い文字列を簡略化するときである。

[name@pc150 ~]$　set d=/home/s01/name/path1/path2/path3/path4
このようにすると，/home/s01/name/path1/path2/path3/path4 という文字列は，以後 $ をつけて$d で参照できる。

[name@pc150 ~]$　cd $d
[name@pc150 ~]$　set.....設定されているシェル変数を表示する。
history 環境変数（history コマンドが遡ることのできる個数）を 25 にセットするには，
[name@pc150 ~]$　set history=25

コマンドラインから入力した設定した環境変数は一時的なものだから，logout するとまた元にもどってしまう。恒久的なものにするには ~/.cshrc をエディタで編集しなければならない。（なお実習用 PC では~/.tcshrc を設定しないこと。設定が混乱すると構造実習に支障をきたすおそれがある。）

6. ワイルド・カード文字（メタ・キャラクタ）
似たようなファイル名（mydatafile1, mydatafile.c, mydatafile9, mydatafile.data ）があるとき，文字列の一部にワイルドカード文字を用いて，特定のファイルを表すようにできる。ワイルドカードとは，カードゲームでの Joker のようなものだ。

*　空文字（文字数ゼロ，または文字がない）を含む，いかなる文字列をも表す。
[name@pc150]$ rm *　........　　すべてのファイル（ピリオドで始まるもの以外）←この

操作をすると必要なものをひっくるめてすべて消えるのでやっちゃだめ！
[name@pc150]$ ls *.c .... 末尾が.cで終わるすべてのファイル
[name@pc150]$ cat a*z ..... aで始まりzで終わるすべてのファイル

? 任意の一文字を表する。
[name@pc150]$ rm mydatafile?.c ...... mydatafile0.c, mydatafile9.c, mydatafilex.c を表す。

［ ］このブラケット内のいずれかの一文字を表する。
[name@pc150]$ ls mydatafile[12].c
.....mydatafile1.c, mydatafile2.c を表す。

［ ］と「ー」（ハイフン）
置き換える「一文字」の範囲を表す。
[name@pc150]$ ls a[A-Z] ..... aA, aB, ..., aZ を表す
[name@pc150]$ ls c[10-23] ... c0, c1, c2, c3 を表して，c10, c11, ... , c23 を表すことはない。
[10-23]は一文字置換なので 1，0，0，2，3 のいずれかを表す。
結局 0，1，2，3 のいずれか一文字のことである）

7. 単語の補完機能
tcsh ではタイピングを減らすために，コマンドや引数（argument，コマンドに続くファイル名など）などの単語を途中までタイピングして，残りを tcsh に補わせることができる。これには tab キーを入力する。taro, hanako, jiro のファイルがあるとき
[name@pc150]$ emacs h<tab>
とすると，h に続いて anako を補ってくれる。いくつかの候補があり一義的に tcsh が決められないとき tcsh はベルをならす。hanako と hikaru があるとき h<tab>を入力するとベルがなるが，続いて a<tab>とすると hanako を補完してくれる。
あまりベルがなってうるさいときは，候補のリストを表示させることもできる。例えば emacs のスペリングが不明なとき，e（em でも OK）のあとに Ctrl-d（control キーをおしながら d を押す）と入力すると，

```
[name@pc150]$   e <Ctrl-d>
    e       ed      egrep   else    end
    endsw   env     eval    exec .......
    emacs endif ......
[name@pc150]$   e
```

と e で始まる単語のリストを表示し，コマンド行はもとの e を表示して待っている。そこでリストをみて e のあとに macs とタイプすれば良い。

8. 入出力のリダイレクトとパイプ，バックグラウンド処理
「＞」 出力のリダイレクト （注：ファイルは上書きされる）
「＜」 入力のリダイレクト
「＞＞」 追記モードでの出力のリダイレクト （注：すでにあるファイルの末尾に追記される）
「｜」 パイプ。中間ファイルを経由せずに、あるコマンドの結果を次のコマンドにおくる。
「＆」 アンパサンド・・・バックグラウンドで処理する。処理が終わらなくてもカーソルが戻ってくる。
「；」 セミコロン。複数のコマンドを逐次実行させるときにこれで区切って一行に書くことができる。
「‘」 逆シングルコーテーションマーク
逆シングルコーテーションマーク内のコマンドを先に解釈して実行し、その結果を因数として次のコマンドの解釈に用いる。
【例】head `which startx`
この例ではまず which startx によってコマンド startx が実は/home/share/bin/startx というファイルであることをまず解釈し、次にその先頭 10 行を head によって表示させている。

§1 を参照して復習してみてほしい。

9. コマンドの登録更新 （rehash）
実行形式のシェルスクリプトやコンパイル後のファイル，すなわちコマンドを新たに作成したり，外からコピーしてきたこれらのファイルを実行するには注意が必要だ。シェルが持っているコマンド名を登録してある hash table を更新しないと，コピー元の同名のファイルが実行されてしまったり，実行不能のエラーがでたりする。rehash（内部）コマンドを実行してこのテーブルを更新する。
下の例は cmd というコマンドがあったとして、それを現在の自分の directory の mycmd にコピーした後、mycmd を登録更新する例である。

```
[name@pc150]$ cp /bin/cmd .
[name@pc150]$ chmod u+x mycmd
[name@pc150]$ rehash
```

10. シェル・スクリプト内で利用できる制御文
シェル・スクリプトは、これまで述べたように

シェルの機能を利用して、シェルに解釈させるコマンドや処理の羅列を書いたファイルのことである。処理を制御するための繰り返しや条件判断を行うためにif文、foreach文、while文などが使える。またコマンドラインから引数を受け取ることもできるが、今回は詳細は省略する。
ただし、科学技術計算（NMR/X-ray/情報生物学）系のプログラムをインストールしたり、それを利用したりする場合は、しばしばこのシェルスクリプトのお世話になるので、実際に使い始めてからあらためて勉強しなおしてほしい。自分で書けるようになれば一番よいが、他人が作成したものが大体どんな動作をしているのかが理解できるようになれば、当面は十分である。

§5　UNIX/Linux の環境設定

いま UNIX/Linux を使っている端末とは別の端末で login すると画面が正常に表示されなかったり，１文字消去が出来ないなどの不都合が起きることがある。また，いままで使っていたコマンドが使えなくなるなんてことも起こるかもしれない。これらは，UNIX/Linux を使う環境を整備することで回避できる場合が多い。ここでは「ドットファイル」ともいわれる，環境設定ファイル「.login」，「.tcshrc」，「.emacs」を簡単に述べる。ドットファイルを編集するときは通常テキストエディタ vi を使う（後述）。
ファイル名の頭にドット（.）が付いているドットファイルは隠しファイルとなっている。ドットファイルは通常ホームディレクトリにつくられる。つぎのように入力するとドットファイルの存在を確認できる。

```
[name@pc150]$ cd [return]
[name@pc150]$ ls –alF
```

「.cshrc」
ユーザがログインするとまず，シェルが起動する。シェルが tcsh ならば「.cshrc」「.tcshrc」の内容が実行される。【注。おそらくライセンスが有償なため Linux では tcsh を csh の代わりに用いている。そのため、tcsh は.tcshrc と.cshrc の両方を初期設定ファイルとみなす】
内容を見るコマンド more を使って

```
[name@pc150]$  more ~/.cshrc
```

シェルをコントロールするシェル変数は set コマンドを，UNIX/Linux で動くアプリケーションをコントロールする環境変数は setenv コマンドを使う。

1. シェル変数

prompt
入力促進のプロンプトは「%」が標準となっているが実習室では[name@pc150]$のように出るはずである。

set prompt="%d(%h)> "

とすると，プロンプトは「%＜現在のディレクトリ＞（コマンド実行番号）＞」となる。

path
UNIX/Linux がコマンドを探しに行く経路をディレクトリで設定する。
例：set path=(. ~/bin /usr/local/bin /usr/contrib/bin /usr/bin /bin)
これが正しく設定されてないと，「そんなコマンドないよ」とシェルにしかられる。

history, savehist
history はコマンドの履歴を最新のものからいくつ遡って保存するかを設定する。savehist はログアウト時にいくつのコマンド履歴を保存するかを指定する。

```
set history = 25
set savehist = 25
```

noclobber
リダイレクションなどでファイルを書き出すときに上書きを禁止する。

```
set noclobber
```

こうしておくと，ファイル aaa がすでに存在するときに

```
[name@pc150]$ echo "Hello, world" > aaa
```

とすると警告がでて，aaa は上書きされない。強制的に上書きするには

```
[name@pc150]$ echo "Hello, world" >! aaa
```

というように「>!」を使用する。

2. 環境変数

環境変数はシェルが起動しているときに蓄えられている変数であるが、そのシェル上で動かすプログラムからも参照できる。これは導入したマシンやプログラムをインストールした場所、システムの種類などに関わらず同じ動作を保障するためにプログラムを設計するときによく用いられる。

```
[name@pc150]$　setenv
```

とすると現在設定してある変数を見ることができる。
環境変数は通常は大文字を使う。
大抵の科学計算プログラムは、それをインストールまたは起動するときに独自の環境変数の設定を要求する。もしどこかから移植したプログラムが、うまくコンパイルできたにもかかわらず動かないときは、環境変数の設定を疑うとよい。

以下は一般的な環境変数の例であるがここでは詳しくは説明しない。

EDITOR
自分が使う標準のエディタを指定して，UNIX/Linux のアプリケーションに知らせる。
setenv EDITOR emacs

PAGER
画面表示をコントローする際に，どのツールを使うか指定する。通常は more になっているが，これを less（less は画面の逆スクロールができる）にするときは，
setenv PAGER less
とする。

その他のコマンドや初期設定ファイル

umask
作成したファイルを他人に読まれない，実行されないようにファイルの許可属性を設定する。
umask 077
とすれば，自分だけが読み書き実行できる属性がファイルに付く。

alias
UNIX/Linux のコマンドは簡潔で短いので打鍵しやすいが，覚えにくい。これを自分流のコマンド名に変更するときは alias で別名登録できる。この内容を「.alias」に書いておいて，source コマンドを実行する。これで alias が設定できる。

```
[name@pc150]$  alias
alias   bye   logout
alias   h   history
aliasla'ls -alF \!* | less'
[name@pc150]$ source ~/.alias
```

.(file)　　自分のホームディレクトリにあるピリオドで始まるファイル
dot-file などと呼ばれる。通常のファイルリストコマンドｌｓでは表示されない(隠しファイル) ls –laF などとしてやると見ることができる。起動時に必要な環境変数の設定や、X-window の設定などのほか netscape の設定ファイルなどが格納される(プログラムが勝手に作成することも多い)

「.login」
ログインして「.cshrc」の内容が実行されると，ついで「.login」の内容が一度だけ実行される。「.cshrc」はマルチプロセスを実行するたび(シェルを起動するたび)に何回でも実行される。

「.emacs」
これはエディタ emacs / Mule の操作属性を設定するファイルで，その内容は emacs lisp というプログラミング言語で記述する。ここでは，システム管理者があらかじめ作成してくれた「.emacs」で十分だろう。

「.Xauthority / .Xdefaults」
X-window 関連の設定ファイル。

§6　通信・ネットワークのコマンド

1. 通信が届いているかどうか調べる　(ping)
[name@pc150]$　ping　(相手先のコンピュータ・ホスト名)　または
[name@pc150]$　ping　ip-address
＜Ctrl-C＞で終了
物理的に通信がつながっている場合には、送ったパケットが届くまでの所要時間が表示される。

2. リモート端末機能　(telnet)
telnet は TELNET プロトコル（通信手順の約束事）に従い，呼び出した相手システムの端末の様に動作させるものだ。TELNET プロトコールに従った相手なら UNIX/Linux システムでなくとも，端末として機能する（UNIX/Linux システムの端末機能をさせるには rlogin がある）。

UNIX/Linux のコマンド待ちの状態からこの telnet を直接起動することもできる。
[name@pc150]$　telnet　(相手先のコンピュータ名、または IP)
このときは，大型計算機のリモート端末と全く同じである。

便利な使い方としては，後述するエディタ emacs の中から telnet を起動させることもできるということである。違いは，emacs を経由すると，emacs が telnet のバッファを作るので，画面への出力がファイルとして保存できる（emacs で C-x C-s または C-x C-w でできる）という利点があることだ。telnet を起動するには，ホスト名を指定する。
[name@pc150]$ emacs[return]
M-x telnet[return] ...　　　(M-x は[Alt-x])
ホスト名を聞いてくるので
pc149[return]
login:メッセージをだして，ユーザーID の入力待ちになる。ここからは，相手先のコンピュータの中のコマンドを実行することになる。
exit コマンドを出して終了すると，telnet は

Connection closed by foreign host.

というメッセージ（後のメッセージは無視する）をだして，emacs のバッファに戻る。いままで，大型機と通信した内容（log）はここにそのまま残っている。これを修正して，自分のプリンタに出せばよい。[Ctrl-x Ctrl-s]で通信結果がファイルにセーブできるので便利である。

3. ネットワークでのファイル転送（ftp）
ftp（file transfer protocol）は相手のシステムとの間でファイル転送を行う。相手は通常は UNIX/Linux システムだが、Windows や Mac も ftp 機能を持っているかもしれない。だから，Mac，Win，SGI などで相互にファイル転送ができる。

ftp の起動
[name@pc150]$ ftp linuxsrv ....実習室のサーバマシン（UNIX/Linux マシン）に接続する
相手側に登録してあれば，ログイン名，パスワードを聞いてくる。なお、Linux の配布などを行っている public な site では誰もが（制限付きで）匿名でログインできる anonymous ftp サイトがある。
受け側のディレクトリを設定する（local change directory）
ftp> lcd ~/data ...pc150 の data ディレクトリに移動
相手側のディレクトリを設定する（change directory）
ftp> cd text ...今いるディレクトリから text というディレクトリに移る
相手のファイル hoge を pc150 に持ってくる
ftp> get hoge[return]
相手先の指定したフォルダ（text）の hoge というファイルを，pc150 の data ディレクトリに転送する。mydata というファイルが同じ名前で自分のところに作られる。
こちら(local)のファイルを相手に持って行く
ftp> put mydata[return]
pc150 の今いるディレクトリ（data）の mydata というファイルを相手先の指定したフォルダ（text）に転送する。
複数のファイルの転送：mput / mget
>ftp mput *[return]
abc.xyz ?...こんなかんじにファイル名*を展開して，ftp が転送の可否を尋ねてくる。オーケーなら y を，スキップするなら n をタイプすると，次のファイルについて尋ねてくる。
ftp を終了する
ftp > quit[return] ...終了する。

§7　応用 UNIX/Linux コマンド

UNIX/Linux で研究のデータ処理をすることの最大の利点は、データの入ったファイルを扱うための便利なコマンドが、システムとしてもともと備わっている点である。たとえばどんなことができるかというと、

並べ替え
ある文字列を含む行の抽出
検索・置換

などである。こうした操作は、Mac や Win の上では通常 Exel などの表計算ソフトで行う。これがコマンドの組み合わせだけで簡単にできるというのが UNIX/Linux の最大のメリットである。

UNIX/Linux で動作するプログラムのデータファイルは大抵はテキスト形式であり、（つまり more /less で内容を読むこともできるし editor で編集もできるということ）、文字や数値がスペースで区切られた複数の行からできている。

そうしたファイルに作用して、いろいろな操作を行うコマンドのうち、使ったら便利と思われるものをあげておく。なお、このような処理を行うために更に一般的で、万能で、しかし少し煩雑な方法としてはプログラミング言語 Perl を使ってスクリプトを書く、というのがある。これについては実習初日の第三部で、Perl 独習などの教材を利用して、あらましを概説するのでこのテキストからは省いた。

1. ファイル内の一致する文字列を検索する（grep / egrep）
この目的のためには， file1 から文字列 abc を検索するには，grep コマンド（global regular expression printer）を使う。

```
[name@pc150]$ grep Ala aa.txt
[name@pc150]$ grep -n Ala aa.txt
[name@pc150]$ grep -v Ala aa.txt
```

-n オプションを使用すると，検索文字列を持つ行の行番号を合わせて表示する。
-v オプションを使用すると，検索文字列を持たない行を表示する。

この grep コマンドは UNIX/Linux の正規表現（regular expression）を使用するともっと多彩な検索が可能になる。

2. ファイル内の特定のパターンを変更する（tr）
tr コマンドは文字列を置き換える（translate）。

入力ファイルはリダイレクションまたはパイプを使って指定する。
つぎの例は，mydatafile 中のすべての “ycu” を “yokogama-cu “に置き換える。

[name@pc150]$ tr “ycu” “yokohama-cu “ < mydatafile
[name@pc150]$ cat mydatafile | tr “ycu” “yokohama-cu “

[ ]を用いると一定範囲の文字を指定できる（このときは［, ］をシェルに正しく解釈させるために引用符”が必要になる）。
次の例はすべての小文字を大文字に置き換える。

[name@pc150]$ tr "[a-z]" "[A-Z]" < mydatafile

次の例は mydatafile で使われているすべての単語の区切り文字（スペース，コンマ，ピリオド，！，？など）を改行文字（\012 は８進数での NEW LINE，改行）に置き換え，アルファベット順に並べ変え，重複行を取り除いた単語帳 wordlist を出力する一連のコマンドだ。-c オプションは変換元の文字（大文字 A～Z，小文字 a～z）に含まれない文字（すなわち区切り文字）を置き換えの対象にすることを示し，-s オプションは変換後の文字（改行文字）が連続するとき１個の文字に短縮することを示す。

[name@pc150]$ tr -cs "[a-z][A-Z]" "[\012*]" < mydatafile | sort | uniq > wordlist

3. ファイル内の一部を編集する（sed）
sed=stream editor
上記 tr によく似ているが、文字の置換以外に削除や行・ファイルの挿入といった高度な編集機能を持ち合わせているので、一連のファイルに決まりきった操作を繰り返すような作業のときに用いる。いくつものファイルを連続して処理しなければいけないときにいちいちエディターを起動しないで済むというのが最大の利点である。**grep** や後述する **awk / nawk / gawk** でも用いられる**正規表現**を解釈することができるので、柔軟性にとむ。

[name@pc150]$ sed ‘s/ycu/yokohama-cu/g’ mydatafile

sed はシングルコーテーションで囲まれた「命令」を mydatafile に施す。
「命令」＝アルファベット１文字 ＋ 「デリミタ」 ＋ パターン１ ＋「デリミタ」 ＋パターン２＋「デリミタ」＋オプション
という形式をしていることが多く、デリミタは[/]のほかに「＃」や[@]なども使える

4. パターン処理言語 awk （オーク）
gawk（Linux） /nawk (SGI, UNIX)
このコマンドは読み込んだファイルの各行に対して、同じ処理を繰り返し行うときに非常に便利なコマンドである。
・ テキスト・パターンを正規表現をもちいて検索することができ
・ 条件判断を行い、
・ 行から特定のフィールドを抜き出し、
・ 整形して出力する
という使い方に利用できる。それゆえパターン処理言語と呼ばれ、実際にはプログラミング可能な報告書作成ツールとして、簡単なフィルターを作るときに多用される。
gawk は指定したパターンにマッチするすべての行に対して特定のアクションを実行する（パターンとアクションのどちらかを省略することもできる）。
[name@pc150]$ gawk ‘命令‘ ファイル名
のように書く。命令をシングルコーテーションでくくるあたり sed に似ている。
[name@pc150]$ gawk -f 命令ファイル ファイル名 （これは命令が複雑な場合シングルコーテーションの中身を別にファイルにして保存しておいてもいいということである）
命令の形式は
パターン(条件判断文) ｛アクション；アクション；アクション；｝
というようになっている。このパターン＋条件判断文の組み合わせを何組でも書くことができる。
awk を非常によく使うのは、**いろいろなデータファイルのフォーマット変換**である。

[name@pc150]$ gawk -F: '{print $9, $1, $3}' mydatafile

この例は，いくつかのフィールドがコロン（：）で区切られた行からなる mydatafile というファイルから，第９，第１，第３フィールドの順にブランクで区切って画面に出力する命令だ。-F オプションを省略するとブランク（空白）による区切りで行が構成されていると解釈される。｛と｝の中のプログラミングが自由にできる。

[name@pc150]$ gawk '$3 ~/Arg/' mydatafile

この例は，いくつかのフィールドがブランクで区切られた行からなる mydatafile というファイルから，第３フィールドが Arg という値をも

つ行全体を表示するものだ。grep コマンドとは，このコマンドでは行のどの位置（フィールド）に Arg があってもその行を表示する点が異なる。gawk のより詳細な使い方はマニュアルを参照されたい。

5. ファイルからの特定フィールドの切り出し cut

ファイルの特定のフィールド（第１フィールドから数える）を抜き出してファイルのシェイプアップをする。
区切り文字(デリミタ)は-d オプションで指定し、抜き出したいフィールドを-f オプションで指定する。出力するときの区切り文字は別途-o オプションで指定する。

[name@pc150]$ cut -d' ' -o, -f3,5 mydatafile

上の例はフィールドの区切りがブランクのファイル（-d オプション）の第３，第５フィールド（-f オプション）を抜き出して，カンマで区切られたファイルを画面に出力（-o オプション）する。
-d の省略は tab が仮定される。
-o を省略すると-d オプションと同じ区切り記号が仮定される。
フィールドの並びは昇順に指定する。降順（逆順）に指定しても必ず出力は昇順となる。（順序を自由に指定したい時は 前述の gawk を使う）。

6. ファイルの内容をソートする（sort）

指定したファイルのすべての行を指定したオプションに基づいて並べ直して画面に出力する。
次の例は行全体を ASCII 順（ブランク，!, ... ,0,1,2, ... ,A,B,C, ... ,a,b,c ... ）にソートするものだ。通常は ASCII 順でのソートが仮定されている。

[name@pc150]$ sort test

特定のフィールドを対象にソートすることもできる。フィールドは+pos1 -pos2 のように，間に空白をはさんで，指定する。+pos1 は pos1 個のフィールドをスキップすることを，-pos2 はソートの対象にするフィールド範囲の最後のフィールドを表す。-pos2 が省略されたときは pos1 スキップしたフィールドから初めて行末までがソートの対象になる。

[name@pc150]$ sort +5 -6 mydatafile
　第６フィールドだけを対象にソート。

[name@pc150]$ sort +4 mydatafile
　第５～行末までを対象にソート。

次の例は各フィールドがただ１つのブランクで区切られたファイルの第３フィールドを数値とみなしてソートするものだ。n は文字列でなく数値によるソートをすることを意味する。ブランクを詰めて桁合わせした数字をソートするときにはこの n フラグを指定しないと予期しない結果がでることがある。

[name@pc150]$ sort +2n -3 test　　-n オプション　＝数字順でのソート

フィールドの区切りがブランク以外のときは-t オプションの後ろに区切り用の文字を指定する。次の例はコンマでくぎられたフィールドの第２フィールドを昇順に，第４フィールドを降順（ r フラグ）にソートするものだ。

[name@pc150]$ sort -t, +1 -2 +3r -4 test　　-t オプション＝区切り（デリミタ）変更

次の例はコロンで区切られた，第３フィールドを対象にしてこのフィールドの先頭のブランクを無視して（-b オプション）ASCII 昇順にソートするものだ（ASCII 順のソートではブランクも文字として認識されるが，数値のソートでは先頭のブランクは無視される）。

[name@pc150]$ sort -b -t: +4 -3 test　　-b オプション　ブランクなどはソートの対象外とする

7. ２つのファイルの違いを調べる（diff）

似たような名前のファイルが２つあるとき，どこが違うのかをしらべるには，片一方のファイルのハードコピーを用意し，もう一方のファイルを cat や p コマンドで画面にだして双方を首引きでみくらべるよりも，diff コマンドを使用したほうが簡単だ。
これは，通常のテキストファイル（意味のある文字列として読めるファイル）を行単位で比較するコマンドだ。異なっている行が出力される。

[name@pc150]$ diff file1 file2
２つのフアイルに差異がないときは何のプロンプトも出ない。差があったときのプロンプトは，次の３つの命令を含んだ（カンマで区切られた）行番号を出力する：

a（add:追加）
d（delete: 削除）

c（change: 変更）

出力された行番号の後ろには，ファイルの中の追加，削除，変更の対象となる行の内容が表示される。file1 の内容を表示するときは，先頭に '<' がつく。file2 の内容を表示するときは，先頭に'>' がつく。diff は，file1 を file2 と同一のものに変換する命令，情報を出力する。ａ，ｄ，ｃの命令より左がわの行番号は file1 のもので，右側の行番号は file2 のものだ。

[name@pc150]$ cat file1
aaaaa
bbbbb
ccccc

[name@pc150]$ cat file2
aaaaa
ccccc

[name@pc150]$ diff file1 file2
2d1 .... file1 を file2 と同じにするには file1 の２行目を削除する必要がある。
< bbbbb ...... file1 の内容（file1 を file2 と同じにするときは不必要）

8. ２つのファイルの共通の行を調べる（comm）

diff コマンドの逆を実行するコマンドだ。共通行（common），または一方のファイルのみに存在する行を探索する。２つのファイルはあらかじめソートされていなければならない。出力は file1 のみにある行，file2 のみにある行，双方のファイルに共通にある行の，３つに段付けされたカラムからなる。

[name@pc150]$ sort file1 > file11; sort file2 > file22 ....ファイルのソート
[name@pc150]$ cat file11
123 456 789
aaa bbb ccc
[name@pc150]$ cat file22
123 456 789
$$$ %%%
[name@pc150]$ comm file11 file22
123 456 789
$$$ %%%
aaa bbb ccc
フラグ１，２，３は対応するカラムの出力を抑制する。
[name@pc150]$ comm -23 file1 file2 ..... file1 にあり file2 にない行を表示。

9. 重複する行を探す（uniq）

隣接する行を検査し，重複する（または重複しない）行をプリントする。このコマンドを実行するには，隣接するそれらの行が隣接していなければならないので，対象となるファイルをあらかじめ sort コマンドでソートしておく必要がある。
[name@pc150]$ uniq -d mydatafile .... 重複した行の中の１行だけをプリントする。
[name@pc150]$ uniq -u mydatafile .... 重複していない行だけをプリントする。

10. ワードカウント（wc） （word count）

語数、文字数、行数などを調べる
ファイルの大きさは ls（list）コマンドでも知ることができるが，wc コマンドは文字数（バイト数）だけではなく，単語数，行数も知ることができる。

[name@pc150]$ wc mydatafile
7 115 123 mydatafile

のように報告する。読み方は「mydatafile は 7 行， 115 語， 123 字の大きさを持つ」となる。また，複数のファイルを指定するとそれらの合計もわかる。

11. 表形式のファイルのカラムごとの結合（join）

これは覚えておいて損のないコマンドである。実際、この操作ができるエディター自体が少なく、emacs を愛用している人のほとんどがこのカラムごとの編集機能を挙げる。（つまり sed や vi ではできない）
file1 と file2 の各行で指定した２つの関係を並列に結合し，画面に出力する。
file1 と file2 は結合する（通常は最初の）フィールドで ASCII 順にソートされていなければならない。
各フィールドは通常ブランク（スペース）で区切られている。
file1 と file2 の内容が次のような例でみてみると，

<file1>：	<file2>：
001 abc 123 #$%	006 2.3 9.0
006 xyz 456 +*=	004 5.0 8.4
004 tya 862 !&/	009 5.6 6.1
001 5.9 5.7	
002 9.9 8.5	

[name@pc150]$ sort file1 > sort1; sort file2 > sort2 あらかじめソートしておき
[name@pc150]$ join -o 1.1 1.2 1.3 2.2 2.3 sort1 sort2

結果は,
001 abc 123 5.9 5.7
004 tya 862 5.0 8.4
006 xyz 456 2.3 9.0
となる。-o オプションで第一ファイルの１列目（1.1），第一ファイルの２列目（1.2），第一ファイルの３列目（1.3），第二ファイルの２列目（2.2），第二ファイルの３列目が出力された。また，第一列がマッチしない行は出力されていない。この join コマンドと同様のことは cut, paste というコマンドを組み合わせれば実現できるが，この join コマンドのほうが容易だ。

12. オンライン計算機 （ｂｃ －１）
このコマンドを打ち込むと、標準入出力から簡単に数式を打ち込んで計算をさせることができる。
変数を代入して覚えさせておくこともできるので便利である。
[name@pc150]$
12+7 [return]
19
25.0**4 [return]
390625.0000
25.0*4 [return]
100.0
a=12 [return]
pi=3.1415 [return]
pi * (a^2) [return]
452.16
quit ←で抜ける

13. デスクトップ計算機 （xcalc）
[name@pc150]$ xcalc [return]
で画面上に計算機が現れる。代入とかは使えないので bc –l を好む人も多い。

14. 複数のファイルをひとつにまとめる まとまったファイルを展開する（tar）
ネットワーク越しにファイルをやり取りする場合に、複数のファイルやディレクトリ構造を含むファイルをやり取りする場合に、そのディレクトリ情報を含んだ形でファイルを転送できると非常に便利である。そういうときに tar コマンドを利用する。なお慣例では tar でまとめたファイルには拡張子.tar を付ける。利用法は
ファイルをまとめる
[name@pc150]$ tar cvf *hogehoge.tar file1 file2 file3....* [return]
[name@pc150]$ tar cvf *hogehoge.tar filename** [return]
[name@pc150]$ tar cvf *hogehoge.tar directory/* [return]

最初の例は file1, file2, file3, ...をひとつのファイル hogehoge.tar にまとめる。次の例では、現在いるディレクトリの filename ではじまるファイル全てを hogehoge.tar にまとめる。最後の例ではディレクトリ名 directory をひとつのファイルにまとめている。

ファイルを展開する
[name@pc150]$ tar xvf *hogehoge.tar* [return]
まとめられたファイル hogehoge.tar を現在のディレクトリに展開する。なお tar コマンドでまとめられたファイルのことを俗語で tar-ball などと呼ぶことがある。

tar ファイルの中身を確認する
[name@pc150]$ tar tvf *hogehoge.tar* [return]

15. ファイルの圧縮と解凍 （gzip / gunzip）
[name@pc150]$ gzip file1 [return]
file1 が圧縮されて file1.gz を生成する。ファイルの形式にもよるがサイズは 6 割くらいになる。自動的に拡張子.gz が付加するのに注目。

[name@pc150]$ gunzip file1.gz [return]
file1.gz が解凍されて file1 になる。

なお前述の tar コマンドはオプション z を付加することで、圧縮解凍とファイルをまとめる操作を同時に行うことができる。
[例]
[name@pc150]$ tar zcvf hogehoge.tar.gz *directory/* [return]
[name@pc150]$ tar zcvf hogehoge.tgz *directory/* [return]
両方ともディレクトリ directory を hogehoge.tar.gz または hogehoge.tgz にまとめている。tar と gzip 両方を施したファイルの拡張子には.tgz を用いることができる。展開方法は同様に
[name@pc150]$ tar zxvf hogehoge.tgz [return]

【ノート】

UNIX/Linux 初級講座・エディター編・
Vi と Emacs/XEmacs

廣明　秀一
hiroakih@tsurumi.yokohama-cu.ac.jp
横浜市立大学大学院総合理学研究科生体超分子システム科学専攻

この章では vi と emacs の二つのエディターの簡便な使い方を実習する(約 1.5 時間)

§１　Vi　（エディターその１）

UNIX のテキストエディタとしてもっともポピュラーな vi の使い方を簡単に紹介する。
vi は「標準エディター」であり、それなりに高機能であり、しかも X-window が起動していない状態(コンソールモードやリモートで telnet などから操作する場合)でも確実に動作する。マウスや矢印キーが無効でも、使うことができる、など利点がある。しかし、その独自の操作性やコマンド体系はなかなか覚えにくく、好き嫌いがはっきりしているエディタであるので、必ずしも覚えなくてもよい　（§１０参照！）Windows 等の普通のスクリーンエディタを使い慣れている人には、使いづらいに違いない。

ただし NMR や X-ray などでは非常にしばしば、測定装置制御用のワークステーション（SGI）に後述の emacs がインストールされていなかったり、SGI 標準のテキストエディタ jot が非 SGI マシンからのリモートでは使えないなど制限が多い。そのような研究に携わる人は、好むと好まざるにかかわらず vi の最低限は知っておいて損はない。

- vi には、「コマンドモード」、「入力モード」が存在し、モードを切り替えながら利用していく。
- 起動直後は「コマンドモード」になっているので、「モード切り替え」を行わない限り入力すらできない！
- 入力モードへの切り替えコマンド（アルファベット一文字）を打って初めて文字の入力が出来るようになる。
- 現在入力モードなのかそうでないのか、見ただけではわからない！[ESC]を押すとコマンドモードに戻るので、とりあえず[ESC]を空うちするクセをつけるとよい。

■起動

vi ファイル名	編集対象のファイルを開く（複数のファイルも可）
vi + ファイル名	最後一画面を表示する
vi -r ファイル名	壊れたファイルをリカバリ

■入力（コマンドモードでこのコマンドを打つと入力モードになります）

a	カーソルの右から入力開始
A	行末から入力開始
i	カーソルの左から入力開始
I	行頭から入力開始
o	現在行の下に１行挿入し、その行頭から入力開始
O	現在行の上に１行挿入し、その行頭から入力開始
[ESC]	入力モードから抜ける

■カーソル移動
基本的には矢印キーで移動すればよいがキーボードの特殊キーが効かない場合など
なお、すでに入力モードに入っているときは矢印キー以外は文字として入力されるので注意

h , BS	１文字左へ移動（←）

j , CTRL+N	１行下へ移動（↓）
k , CTRL+P	１行上へ移動（↑）
l , SPACE	１文字右に移動（→）
H	画面の最上行に移動
M	画面の中央行に移動
L	画面の最下行に移動
G	ファイルの最終行に移動
*n*G	ファイルの n 行目に移動

■削除

x	カーソル上の 1 文字削除
*n*x	カーソル上から *n* 文字削除
X	カーソルの左の文字を 1 文字削除
dd	現在行を削除（バッファにコピーされる）
*n*dd	n 行分削除
dw	カーソル上の一語を削除
*n*dw	カーソル上の *n* 語を削除
df 字	カーソル位置から指定した字までを削除
d$	カーソル位置から行の最後までを削除
d^	カーソル位置から行の先頭までを削除

■置換

r	カーソル上の 1 文字を他の 1 文字に置換
R	カーソル上の文字から ESC が押されるまでの文字列を置換
s	カーソルのある 1 文字を他の文字列で置換
S	現在行を他の文字列で置換
cw	カーソル位置からこの語の最後までを置換
cf 字	カーソル位置から指定した字までを置換
C	カーソル位置から行の最後までを置換

■カット＆ペースト

yy	現在行をバッファにコピー
*n*yy	n 行分をバッファにコピー
yw	単語をバッファにコピー
p	バッファ内のテキストを挿入（文字、単語はカーソルの右に、行は現在行の下に挿入される）
P	バッファ内のテキストを挿入（挿入位置はカーソルの左、または現在行の上に）

■ファイル操作

:w	現在のファイルに保存
:w ファイル名	指定ファイルに保存
:w! [ファイル名]	書き込みを強行
:行 1, 行 2 ファイル名	行 1 から行 2 のテキストをファイルに保存
:w >> ファイル名	現行ファイルの最後に書き加える
:r ファイル名	現在行の次の行にファイルを読み込み、挿入する
:r	現在行の次の行に現在のファイルを読み込み、挿入する
:n	複数個のファイル編集時、次のファイルを編集対象とする
:args	編集ファイルの一覧を表示する
:e ファイル名	指定ファイルを編集対象とする
:e#	一つ前の編集ファイルに戻る

■終了

ZZ	vi を終了（内容が変更されている場合は保存）
:wq	ファイルに保存して vi 終了
:q	vi を終了（内容が変更されている場合は警告）
:q!	vi の強制終了（内容が変更されていても保存されない）

よく使う Vi 機能、早見表

1.	vi		vi を起動する
2.	[shift]ZZ	シフトゼットゼット	ファイルを保存して終了
3.	:q!	コロンキュービックリ	ファイル保存しないで終了
4.	[ESC]	エスケープキー	入力モードから抜ける
5.	i	アイ(insert)	挿入モード入力
6.	a	エー(addition)	付加モード入力
7.	o	オー(open)	新しい行を開いて入力
8.	x	エックス	一字削除
9.	dd	ディーディー	一行削除
10.	r	アール	一字を置換
11.	cw	シーダブリュー	一語を置換
12.	u	ユー	アンドゥ(直前の操作の取り消し)

---

§２　Emacs　（エディターその２）

UNIX/Linux で使われているもう一つのエディター emacs の使い方について書いておく。ひとくちに emacs といっても、操作性のほとんど同じエディターが何種類も一つのマシンに入っていることがある。emacs, xemacs, mule, jed などである。ボタンやマウスによる操作がしやすくなっている改良型 emacs や xemacs のほうが使いやすいかもしれない。
emacs はテキストの入力，編集といった単純なエディタ（editor）にではない。電子メール，文献検索，蔵書検索, telnet を利用した remote アクセスなどにも日常的に用いることができる。UNIX のほとんどの作業は emacs の中から実行可能である。
UNIX システムではフルスクリーン・エディタである vi が標準であり、emacs はどの UNIX でも備えているとは限らない。（注 ： これは歴史的に HDD やメモリの物理的上限がタイトだった時代には、emacs は高機能すぎてリソースに負担をかけていたからである。現に、Bruker 社 NMR を制御する SGI ワークステーションには、未だに emacs は導入されていない。）　しかし一方で、とくにプログラミングを行う人を中心に、vi に代わって標準エディタとして使う人が増えてきた。これは特に C プログラミングの書式にあわせた自動整形機能（筆者は設定の仕方を知らないが）が生産性を大きく向上させるからである。実際に昨年度の実習では学生に C や Perl のスクリプトを、短時間にかなりいろいろ作成したり入力したりしてもらったが、**XEmacs で入力していた学生には打ち間違いやエラーが有意に少なかった。**これは自動段下げ機能や、開き括弧・閉じ括弧・引用符の対応関係の自動チェック機能などが有効に機能した証拠である。
なお、以下にはアイコンボタンやツールバー、マウスなどを用いないショートカットの例をあげたが、このようなコマンドを一切覚えなくとも、現在の Linux に標準装備されている Emacs / XEmacs はボタンとメニュー操作だけでほとんどの機能が利用できる。従って、実習では特に解説は行わないので、興味のある人は各自独学して使いこなして欲しい。
emacs の日本語の参考文献をいくつかあげておくと：

> *大木敦雄著：入門 Emacs。アスキー出版局, 1994, \1800*
> *伊藤他：mule / vi スーパーレファレンス。ソフトバンク，2000, \2200*

■ emacs 基本操作でよく使われるキー入力

emacs は[CTRL]キーと[ESC]キーを多用して、コマンドモードを切り替えていく（もちろん、マウスを使ってメニューからでも操作はできるが）。そこで、他の文献にならって、１この文書中でのキー入力を表す約束ごとを以下のように定めておく。

- ＜文字＞ ...１文字押す。＜文字＞は大文字・小文字を区別しない。
- Ｃ－＜文字＞ ...コントロール・キー（control）を押しながら，＜文字＞キーを押す。１回のキーボード操作である。例えば[CTRL][M]を単に C-M と書く。
- Ｍ－＜文字＞ ...エスケープ・キー（ESC）を押してからいったん離したのち，ついで＜文字＞キーを押す。
- [return] ...return キーを押す。たいてい，[return]は文字列の後ろに必要となる。＜文字＞の後ろはたいてい不要だ（たとえば，C-x 等に[return]はいらない）。
- <tab> ...tab キーを押す。
- <delete> ...delete キーを押す。
- <space> ...スペース・バーを押す。

■ emacs の起動

[name@pc150]$ emacs[return]

または

[name@pc150]$ emacs ファイル名[return]

既にファイルが存在するときは，そのファイルを編集することになるが，指定したファイルがないときはそれを新規に作成することを意味する。ファイル名を誤って入力したときもシステムは新規作成と解釈するので注意しよう。

起動直後の画面にメッセージやヘルプが表示される。emacs のチュートリアルの起動方法があるので、ぜひ試して欲しい。

■ emacs の終了　　　　　C-x C-c　（close）。

画面の最下行で何かを聞かれたら，とりあえず y (yes)と答える。

■ チュートリアル起動　　　C-h T

コントロールキー（control）を押しながらｈ（大文字，小文字を問わない）を押し，ついで大文字のＴを押す（小文字のｔだと英語の tutorial が起動する）。[return]はいらない。以下同じなので表記は簡略化する。

■ コマンドのエラーの回復　C-g

入力途中の文字列や，コマンドの実行を取り止める

コマンドを投入するとベルがなることがある。このようなときは入力コマンドが間違っていたときだ。回復はつぎのようにする。delete キーや BS キーでは取り消しできない。

■ 削除コマンド

■ 一文字消去　　　　　　C-d

カーソル上の（反転している）文字を消す（delete）。

■ 一文字消去　　　　　　delete キー

カーソルの左側の文字を消す。

注　：　似ているキーに BS (back space)があるが、ヘルプメニューが表示されて消去されない。間違えないように。

■ 行末まで削除　　　　　C-k

現在カーソルのある位置から行末までの文字列を一挙に消す（kill）。

■ ＵＮＤＯ　　　　　　　C-x u

誤って消した文字を復活させる。

ヒストリーが設定されており、前回の変更を遡って復旧させることができる。

■ カーソル移動　　矢印キー、その他

つぎのキー操作でカーソルを上，左，下，右に一文字づつ移動できる。押し続けるとリピート機能が働く。

←　（C-b）　カーソルを１文字前に移動（backward）
→　（C-f）　カーソルを１文字後ろに移動（forward）
↓（C-n）　カーソルを１行後ろに移動（next）
↑（C-p）　カーソルを１行前に移動（previous）

（カーソルを行頭，行末に移動）
C-a　カーソルを行頭に移動（ahead）
C-e　カーソルを行末に移動（end）

（画面単位で移動・ページング）
C-v　カーソルを１画面後ろ（次の画面）に移動（Λ）。
M-v　カーソルを１画面前（次の画面）に移動（V）。
（ESC キーを押して，いったん離してから v キーを押す）

（ファイルの先頭，末尾に一挙に移動）
C-<　カーソルをファイルの先頭に移動
C->　カーソルをファイルの末尾に移動

■ 文字列を検索する　　C-s　(...) <ESC>

emacs での文字列の検索は，ファイルの特定の文字列をみつけてそこにカーソルを移動させる。
emacs の検索の特徴は incremental search だ。この検索法では，検索単語を構成する文字を入力する度にそれまでの文字列を探し，カーソルがダイナミックに移動する。
注意：検索では，通常は，大文字，小文字は区別されない。

C-s　incremental search の開始。
画面最下行に，I-search: と表示される。

続いて，
h ...h を入力すると，すぐ h を探して，その後にカーソルが移動。
ha ...a を入力すると，すぐ ha を探して，その後にカーソルが移動。
han ...n を入力すると，すぐ han を探して，その後にカーソルが移動。

<ESC>　ESC キーを入力すると，そこで検索ストップとなる。
<delete>　delete キーを入力すると，後ろから１文字が消え，検索文字列の修正ができる。

■ 再検索　　C-s

上記の han が求めるものでなく，次の han の検索に進みたいときは，再び C-s を入力する。

■ 逆方向検索　　C-r

逆方向（ファイルの始めの方向にむかっての）検索（reverse）。
また，通りすぎた han を検索したければ，C-r とすればよい。

■ 一括検索(順方向)　　C-s　<ESC>　(...)

前期の incremental search では画面がインタラクティブに変化するので，チラチラしてうるさい。
通常の検索（一括型検索）もできる。<ESC>を押す順番に注意。

C-s <ESC> hanako [return] ...control キーを押しながら s を押し（画面最下行に I-search: を表示），ついで ESC キーを押し（画面最下行は Search: に代る），hanako と入力して，return を押す。

■ 一括検索（逆方向）　C-r <ESC> hanako [return]

■ 文字列の置換　M-%　word1 [return] word2 [return]
emacs での置換は Query Replace（利用者に質問しながらの置換）が基本。

M-%　最下行に Query relpace: と表示されるので，
word1[return]　最下行に Query relpace word1 with: と表示されるので，
word2[return]　カーソルは最初の word1 へ移動して，最下行に Query replace word1 with word2: と表示される。確定かスキップを選択する。
M-% 以後の操作
<space>　置換を実行
<delete>　置換せずに，次の文字列に移動
.（ピリオド）　置換して，処理を中止
<ESC>　置換せずに，処理を中止

■ 領域の削除と移動（cut & paste）
■ カット　C-<space>（カーソル移動）C-w
C-<space>に続いてカーソルを移動し，マークから現在のカーソル位置までをカット。画面からこの領域が消える。

■ コピー　C-<space>（カーソル移動）M-w
C-<space>に続いてカーソルを移動し，マークから現在のカーソル位置までの領域をコピーする。画面からはこの領域は消えない。

■ ペースト　C-y
カーソルを任意の場所に移動して，ペースト（yank）。

このカット・アンド・ペーストを矩形の領域に対しても実行することができる。詳しくは日本語マニュアルを見よ。

■ マルチプル・バッファとマルチプル・ウィンドウ
emacs を使って作業しているテキストは，バッファと呼ばれるところに置かれている。emacs にファイル名を指定して，起動するとそのファイル名のバッファがつくられる。ファイル名なしで起動すると，*scratch* というバッファがつくられる。通常はこのバッファは 1 画面を使って表示される。使用者は新しいウインドウをひらいたり、バッファ，ウィンドウを作成，消去，選択できる。これと先ほどのカット・アンド・ペーストを組み合わせて，複数ファイルの相互の編集ができる。

■ バッファの操作
C-x C-b　現在あるバッファのリストを表示する。
C-k k　バッファの名前を指定して，削除する。
C-x b buffer-name[return]　buffer-name にバッファ名を指定して，選択。

■ ウィンドウの操作
C-x 2　選択したウィンドウを上下の２つにわける。
C-x 5　選択したウィンドウを左右の２つにわける。
C-x o　他のウィンドウに移る
（数字の０ゼロではなく other の英字の o）
C-x 1　選択したウィンドウ以外のウィンドウを消す。
画面全体を１つのウィンドウにする。

（1 は数字の 1 で英字のエルではないことに注意）

- ファイルの保存　C-x　C-s　(save)
新規作成（ファイル名なしで emacs を起動したとき）の時は，保存するファイル名を聞いてくる。格納するディレクトリに注意して，ファイル名を入力する。
　既存のファイルの編集のときは，その名前のまま保存する（save）。
C-x C-w　別の名前で保存する（write）。

- ファイルの呼びだし　C-x C-f　ファイル名　[return]（find）
起動時にファイル名を入力しなかった場合にはこのコマンドでファイルを開く。

emacs は「ファイル名」のファイルを探し，あればバッファを作りそこに呼び込む。
なければ，新規にバッファ「ファイル名」を作る。

- ファイルの呼びだし 2　C-x d　[return]（find in the directory）
起動時にファイル名を入力しなかった場合でいまいるディレクトリを探したいときはこのコマンドでディレクトリのウィンドウが開くので，そこからファイル名を選ぶことができる。

- ファイル挿入　C-x i ファイル名[return]　(insert)
現在のバッファのカーソル位置にファイルを挿入する。C-x C-i ではないので，要注意！

- バッファの印刷　M-x lpr-buffer
emacs のバッファが複数のバッファを表示している場合で、バッファ名を指定して印刷することが，上記のコマンドでできる。

- trr の起動　M-x trr
trr が起動する.

---

§3　その他　エディター

UNIX/Linux を日常的に使用して研究に役立てようというユーザーは、なるべく早くに vi か emacs かあるいはその両方を、過不足なく使えてコンピュータ上のファイルの編集を効率よく行うことができるようになるまで慣れる必要がある。しかし、初めての人には機能が豊富すぎて、しかもコマンドが難解なため、どちらもなかなかとっつきにくいであろう。単にちょっとしたファイルを入力したり、編集してセーブするだけ、ということであれば、もっと直感的に扱えるテキストエディターがないと、この情報科学実習の２日目以降が成立しなくなる恐れがある。そこで MacOS でいうところの SimpleText, Windows でいうところのメモ帳　に該当するような、わりと簡単につかえそうなエディターを実習室の環境に実装しておいた。

- vedit
- gedit
- xedit

の３つである。
初日の講義で「vi はキライ」とか「どうも自分は emacs が使えそうな気がしない」とかいう場合は、躊躇なくどちらか使いやすいほうを試してほしい。

なお SGI ワークステーションでは他に「jot」が標準装備されており、これは Macintosh の SimpleText に近い操作感覚の、検索機能などもある便利なエディターである。
また、大抵の UNIX / Linux プラットフォームにはフリーウェアの NEdit を入れることもできる。

Perl を利用した簡単なプログラミング入門

廣明　秀一
hiroakih@tsurumi.yokohama-cu.ac.jp
横浜市立大学大学院総合理学研究科生体超分子システム科学専攻
生体超分子計測科学研究室　助教授

目次

0. perl とは何か？
1. perl の 実行の方法
2. 入力 と 出力, 暗黙の変数 $_
3. 変数、変数の結合、変数の分離 split
4. 四則計算、初等関数、文字変数の計算 join
5. 文字変数の計算・応用編・置換と検索
6. 制御文ことはじめ　条件分岐とコマンドラインからのパラメータ受け取り@ARGV
7. その他の制御構造 for, while
8. ファイル入出力, open close, Unix/LINUX コマンドとの連携
9. 大きなプログラムのつくりかた。サブルーチン sub, ライブラリ

0.Perl とはなにか？

Perl (パール) は、ファイルを読み、書式の自由な変換をして出力することを意図として作られた言語である。
Perl を１００分で速習するのに最適のドキュメントが電通大にあったので、一昨年はそれをテキストとして学習した。今年度はそれを一部改変してテキストに利用する。（わからないところがあったら該当サイトを直接ウェブ上で参照しながら進めてもかまわない。）
http://flex.ee.uec.ac.jp/texi/perl-nyuumon/
上記のサイトはこのテキストのもととなったオンラインテキストである。(10 分 × 10 章 = 100 分 で一応がわかるように設計されている、よい入門サイトであるので、実習終了後もぜひ活用してほしい。ただし上記のオンラインテキストは LaTex や日本語変換などの取り扱いを例としているため、このテキストはむしろアミノ酸配列など BioInformatics 寄りの例題に改変してある。)

0.0　この実習のすすめかた。

　最初に簡単な説明を行う。その後、まずテキスト中の【課題】を６０分でできる限り自習して Perl についての大まかな感覚をつかんでほしい。できれば６章くらいまで各自で進めてほしい。その後質問を受け付ける。最後に、【実習用の例題】に取り組んでもらう。

0.1　Perl には何が出来るか？

たとえば我々は研究の現場で Perl をどのように使っているのであろうか？一例をあげると、MolScript の使用法のところで、.mol ファイルを読みながら条件に応じて色を指定していく作業があった。たとえば学会発表や論文の締め切り間際に、自分のテーマに関連する蛋白質の立体構造を 10 個、全て同じ色調のリボン図にしたい場合など、果たして手作業でちまちまとやっている余裕などあるだろうか？
Perl（や他の Linux コマンド・gawk/sed/grep）が即座に瞬間的に使えるかどうか、というのはそういうときにこそ威力を発揮する。この実習を受けている皆さんは、「今」だったら Perl を学ぶ余力があるだろうが、同じ余裕が１年半後にはないかもしれない。また、Perl はインターネットのサーバーでも、具体的には cgi プログラムという形で非常によく利用されている（たとえばチャット、掲示板、メール送受信スクリプトなど）。覚えておいたら損はない。
Perl 言語の特徴は

1. インタプリタである。(コンパイル不要である。そのせいで実行速度は遅い)。
2. 単純作業を繰り返しさせるのに向いている。
3. テキスト形式のファイルや文字列を取り扱うのにむいている。（Linux の「拡張正規表現」といわれる文字列取り扱いのための演算子が完備）
4. プログラムの形式が C や Fortran のように厳密ではないので、柔軟に書きやすい
5. さまざまな省略表現が可能である。（だがそのせいで、他人の書いたスクリプトはすごく読みにくい）

1.perl の 実行の方法

1.1 Perl の file の 作り方

perl の file は、おしりに .pl という 識別子をつける。file は、~/bin という directory をつくって自分の好きなエディタで作成する。

【例題 1.1】以下の文を p1-1.pl に マウス で copy し、p1-1.pl に save せよ。

```
#!/usr/bin/perl
print "hello world \n"; # this prints "hello world".
```

Perl スクリプトのきまり・その１

1. 最初の行は必ず必要。この file が perl を用いて実行されることを示す。
   (次の行は、hello world を 印刷する。)
2. ＃以下、その行のおわりまでがコメントとなるので実行には、関係がない。
3. 文の終わりには必ずセミコロンをつける。
4. \n が 改行を 意味する。
5. 文字の定数は "" でくくる。

perl の 文の終りには セミコロン；をつけなければならない。エラーとなる。セミコロン がないと、Perl は次の行に文章が 続くと期待し、2 つの文章がつながったものを解釈する のでエラーになる。もしエラー が発生したら、セミコロン；が適切についていることを確認すれば、多くのエラーは回避できる。

1.2 Perl の file を実行するには?

p1-1.pl は、このままでは実行されないので Unix/LINUX の コマンド chmod を用いて実行可能形式にする。

```
% ls -l p1-1.pl
-rw-r--r--  1 rsaito        89 Sep  6 11:13 p1-1.pl
        #file が あることを確認する。
% chmod 755 p1-1.pl
        #実行可能にする。
% ls -l p1-1.pl
-rwxr-xr-x  1 rsaito        89 Sep  6 11:13 p1-1.pl*
        #実行可能であることを確認する。
```

x がついていれば実行可能。ここで file 名をコマンドとしていれると実行する。

```
% p1-1.pl [enter]
hello world
%
```

【例題 1.2】
p1-1.pl で \n がある場合と、ない場合をつくって実行せよ。また、

```
hello
world
```

と印刷するにはどのようにしたらよいか?

解答例:

```
#!/usr/bin/perl
print "hello\nworld \n";
        # hello world を 2 行に印刷する。
```

2.　Perl の入力と出力、「暗黙の変数」
2.0 1 行だけの入力と出力

Unix/LINUX の標準入力から入力するには < >（左不等号と右不等号）を用いる。標準入力・出力とは Unix/LINUX の言葉で、keyboard からの入力と画面への出力と考えていてください。以前説明したように、Linux の便利なリダイレクト機能により、【画面に出力できるものは必ずファイルに出力できる】ので、ともかくカンタンにプログラムを書きたいときは、出力を画面に出すものをつくってしまえば当面の目的が達成できるのだ。

【例題 2.0】　標準入力から 1 行読んでそれを 標準出力に表示する。(p2-0.pl)

```
#!/usr/bin/perl
$a = <>;
        # 標準入力から 1 行読み 変数 $a に代入する。
print $a;
        # $a を印刷する。
```

ここで $a は 文字変数だ。p2-0.pl を save し実行する。

```
% chmod +x p2-0.pl          <== 実行可能にする。
% p2-0.pl                   <== 実行する。
test                        <== test と打ち込む。
test                        <== 標準出力 に test
        と表示された。
%
```

2.1 複数行の入力と出力

入力がなくなるまで繰り返し入力をして作業するには、【while(< >)｛すること;｝】という書き方を用いて繰り返させる。{すること;} のところに入力 がなくなるまで繰り返し作業するための処理内容を書けばよい。ちなみに {...} のあとには；は不要だ（不要だがセミコロンは単独で存在してもエラーにはならないので、私は行の終わりには必ずセミコロンを入れておいている）。

【例題 2.1】　標準入力から入力がなくなるまで 1 行づつ読んでそれを標準出力に表示する。(p2-1.pl)

```
#!/usr/bin/perl
while(<>) {          # ①
print;               # ②
}                    # ③}
```

①　　毎回（）内の条件判断をまず行い、{から} までの間の処理をその条件が満たされなくなるまで繰り返す。<>は標準入力から 1 行読み込む、という命令と、もし 1 行が読めなかったら「条件は偽となる」という処理を兼ねている。
②　　標準入力から読み込んだ内容を出力
③　　while 処理の終わりを示すカッコ

p2-1.pl を実行してみるとわかるが、入力をすると同じものが 印刷される。「入力が終了」するためには、EOF 記号, End of File、キーボードから [Ctrl][d]を入力する。通常の file の終りには自動的に EOF がついている。

キーボード入力の代わりに file として、p2-1.dat という file をつくってそこに、適当な内容を書き込む。標準入力かわりに、 Unix/LINUX のリダイレクション を使って、

```
% p2-1.pl < p2-1.dat
```

とすると、キーボード入力の代わりに p2-1.dat の内容が入力される。これは Linux コマンドで

```
% cat p2-1.dat
```

とやった場合とほとんど一緒である。

2.2 暗黙の変数 $_　暗黙の配列変数 @_

Perl を使う上で、もっとも便利な変数、そしてもっとも不便でありもっとも理解しにくい変数が、この「暗黙の変数」$_　（ドル記号とアンダーバー）である！　ということは、逆にいうとこれさえ理解マスターしてしまえば怖いものはほとんどないのである。心してとりかかるように！
なお、「変数」がなんなのかピンと来ない人は 2.3 / 3.0 のほうも読みすすめて欲しい。
p2-1.pl の②で print; として、print するものがなにも書いていないのに、 出力されるのが不思議に感じられた人は、なかなか鋭い。
Perl では、<u>暗黙の変数 $_</u> というものがよく使われる。ここで print; とは暗黙の変数を印刷する、即ち print $_; のことなのだ。では $_がどこで設定されるかというと、p2-1.pl の場合には while(<>)で入力が存在した場合に、while で $_ が設定されるのだ (これは while 文だけの特別な省略法である)。

while(<>)
を 省略しないで書くと、
while($_ = <>)
(標 準入力 <> を $_ に代入することが真(代入できる) である限り) になる。ちなみに、Perl では=は代入記号である。式の両者が等しいことを条件判断する場合には==を使う。

Perl で使用される他動詞となるほとんどのコマンド (print, shift 等) で目的語(コマンドが働く対象)となる変数が省略されている場合には、常に暗黙の変数 $_ がある(省略されている)と理解してほしい。これは、普段から論理的に曖昧な言語を使っている日本人には向いているかもしれない。なれてくると、通常の言葉(例えば「買ってきました。」とか、「印刷しています。」) と同じで省略した方が便利な場合が多い。
なお、暗黙の変数$_の兄貴分として暗黙の配列変数 @_があることも付記しておく。

【例題 2.2】　p2-1.pl で 暗黙の変数をいれて、同じ様に動作することを確認せよ。

【例題 2.3】　p2-1.pl で 暗黙の変数$_を用いたくなかったら、同じ様に動作するためにはどうしたらよいか？変数$a を使って確認せよ。

2.3　Perl スクリプトのアウトライン

Perl スクリプトのきまり・その２

1. Perl スクリプトは①先頭行と②セミコロンで区切られた文でなっている。
2. 文は大雑把にいって制御文(if / while / for)とその<u>他の命令文</u>からなっている。
3. 制御文には「条件判断式」が必要である。
4. 文の構成要素としては他に
   変数　　　　　$で始まる
   配列　　　　　@で始まる
   配列要素　　　$で始まり添え字が[数字]でつく
   連想配列　　　$で始まり添え字が{}でつく
   サブルーチン &で始まる、ユーザー定義関数
   文字定数　　　ダブルコーテーションのなかみ
   ハンドル　　　すべて大文字で記述
   と
   演算子(x,+,/,-,その他)がある。
5. つまり逆にいうと$や@が付いていないものはすべて、制御文かその他の命令文ということである。なお sin とか sqrt のような算術関数も、「その他の命令文」である。実は Perl ではサブルーチンも含めて全ての命令文は「関数」の形式で定義されている。
6. 関数とは( )で挟まれた引数を受け取って、何か動作を行った後、何かの値を返すモノのことで

ある。print や open も動作の可否にしたがって一定の値を返す関数である。

7. C や Fortran のプログラミングを行った経験のある人ならば気付いているだろうが、**変数に文字か整数か小数か倍精度か、といった区別**はない。前後の文脈に応じて適当に扱われる。

3. 変数・変数の結合・変数の分離 (split)

3.0 変数

先にも述べたが、変数には、$が頭につく。2.2 で暗黙の変数 $_ もその仲間である。変数が順番どおりに並んだものを配列と呼ぶ。配列の要素にも頭に$がつく。配列全体を一度に取り扱う時(配列リテラルと呼ぶ)は同じ記号に@をつける。

【例題 3.0】 変数の代入の例を示す。(p3-0.pl)

```
#!/usr/bin/perl
$x = 0.1;            # 変数 x に数値 0.1 を代入
$m = 'a b c';        # 変数 m に文字 a b c を代入
$y = "x = $x";       # 変数 y に x = 0.1 を代入
$z = 'x = $x';       # 変数 z に x = $x を代入
print "$x, $m, $y, $z \n";
                     # それぞれの変数の値を表示
print ' $x, $m, $y, $z \n ' ;
                     #引用符の中味がそのまま。
```

文字列の代入では、'...' か "..." が使われるが、後者の場合には、 $x 等の変数は変数として解釈されて実際の値が入る。シングルコーテーションでくくった場合には、「$x」という文字のまま打ち出される。

3.1 変数の結合

2 つの変数を結合するには、"$a $b" を用いるか . を用いる（ . はピリオド。見落としやすいので要注意！）。

【例題 3.1】 変数の代入の例を示す。(p3-1.pl)

```
#!/usr/bin/perl
$x = 0.1;            # 変数 x に数値 0.1 を代入
$m = 'a b c';        # 変数 m に文字 a b c を代入
$n = '27 28';        # 変数 n に数値 27 28 を代入
$o = $x.$m.$n;       # $x と $m と $n を結合して
                     変数 o に代入（ピリオド）
print "$o \n";       # 変数 o を表示
$o = "$m $n" ;       # $m と $n を結合するとき間に
                     space を入れる。
print "$o \n";       # 変数 o を表示
$o = $m.' '.$n;      # この形でもよい。
print "$o \n";       # 変数 o を表示
```

おわかりいただけただろうか？

文字変数と数字の変数は適当に文字変数として結合される。それを次の操作で分離すると再び数字に戻る。文字変数の間は四則演算が出来ないが数字にすると演算が出来るようになる。

3.2 変数の分離　split

2 つの変数を分離するには、関数 split を用いる。

【例題 3.2】 変数の分離の例を示す。(p3-2.pl)

```
#!/usr/bin/perl
$_ = 'a b c d e';    #①
split;               #②
print "$_[1] \n";    #③
 print "$_[0] \n";   #④
print "@_ \n";       #⑤
$x = 'x,y,z,2.3,3.2';          #⑥
@x=split(/,/ ,$x);   #⑦
print "$x[1] \n";    #⑧
print "$x[4] \n";    #⑨
print "@x \n";       #⑩
```

<u>解説</u>

① 暗黙の変数 $_ に文字 a b c d e をスペースで区切って代入

② 暗黙の変数 $_ の分割を行なう。

③ 分割した変数の第 1 成分 $_[1] を表示

④ 分割した変数の第 0 成分 $_[0] を表示

⑤ <u>分割した変数の集合である暗黙の</u>配列 @_ を表示

⑥ 変数 $x に文字 a,b,c,d,e を代入

⑦ 変数 $x の , による分割を行ない 配列 @x に代入

⑧ 分割した変数の第 1 成分 $x[1] を表示

⑨ 分割した変数の第 4 成分 $x[4] を表示

⑩ 分割した変数の集合である配列 <u>@x</u> を表示

一番目の split の使い方は暗黙の変数を用いた場合である。2 番目の split の使い方は、コンマ , で $x 区切りその結果を<u>配列@x</u> に代入したものである。何で区切るかを /../ ではさんで示す（これをパターンと呼ぶ）。配列とは、編集の集合 で 0 から 順に成分を持つものである(ベクトルのようなものである。)。配列の成分を参照する場合には、@ ではなく $ を使う点に 注意したい。

split( )は上のように、任意の区切り文字でデータを区切ることができる。パターンには区切り文字を指定できるほか、正規表現に定義されている特殊文字なども利用できるので、非常に柔軟な使い方が可能である。詳細は man perlreg を参照。

【例題 3.3】　次の例は、表の形式のデータを読み込んで その一部(2 番目と 3 番目の データ)を HTML の表形式で 出力するプログラ ムだ。(p3-3.pl)

```
#!/usr/bin/perl
print "<html><body><table>\n";     #①
while(<>) {                         #②
split;                              #③
print "<tr><td>"; #①
print "$_[1] <td>$_[2]  \n";        #④
}                                   #②
print "</table></html></body>\n";
            #①。
```

これに以下の様なデータを p3-3.dat に作る。

```
%vi p3-3.dat
1 2 3 4 5
2 3 4 5 6
3 4 5 6 7
4 5 6 7 8
```

解説

① HTML 形式のファイルにするためのヘッダと表のためのタグを出力する。

② 標準入力がなくなるまで繰り返す while 文（｛から｝まで。）

③ 毎回読み込まれた内容は、暗黙の変数 $_ にはいる。それを分割する。

④ 2 番目と 3 番目の データ を LaTeX の 表入力の内容を印刷する。（添え字は 0 からはじまる）

それでは実行してみよう。p3-3.dat を読み込み p3-3.pl を実行し、p3-3.html に出力する。

```
% p3-3.pl < p3-3.dat > p3-3.html
% cat p3-3.html
(内容省略)
% netscape p3-3.html
```

ブラウザで表になっていたであろうか？

---

4. 計算・関数・文字変数の計算・join

4.0 四則計算

数値変数では、四則計算ができる。ここで用いられる式の=(等号)は数学の等号ではなく代入命令である。つまり、$i = $i + 1;
という式は、$i という 変数に 1 を加えて 結果を左辺の $i に代入せよという意味だ。左辺と右辺が等しいという意味ではない。
$i = $i + 1; は、$i += 1 と省略できる。
(同様に $i *= 2 とは 2 倍するという意味 ($i = $i * 2) だ)。
+= とは 左辺の変 数に右辺の数値を加え、$i に代入するという意味だ。加える量 が 1 の 場合には、さらに $i ++ という表現でも表せる。

【例題 4.0】　四則計算や初等関数の計算の例を示す。(p4-0.pl)

```
#!/usr/bin/perl
#
print 1+2,"\n";                                  #①

$x = 3; $y = 7; print $x+$y,"\n";                #②
$d = $x - $y; print $d,"\n";                     #③
print $x=$x+1; print " ",$x,"\n";                #④
print --$x; print " ",$x,"\n";                   #⑤
print $x++; print " ",$x,"\n";                   #⑥
print '$x * $y'," $x*$y=",$x*$y,"\n";            #⑦
print '$x / $y'," $x/$y=",$x/$y,"\n";            #⑧
print '$x % $y'," $x%$y=",$x%$y,"\n";            #⑨
print '$x ** 2'," $x**2=",$x**2,"\n";            #⑩
```

解説

① 1+2 の結果を表示。この場合には 1+2 が print より先に行なわれる。

② $x に 3 を、$y に 7 を代入して四則計算の結果を示す。和 $x+$y を表示。

③ 差を 変数 $d に代入してから表示す。

④ $x に 1 が加わり 4 になった。

⑤ $x から 1 を引き 表示す。

⑥ $x を印刷してから $x に 1 が加えます。

⑦ $x と $y の積。' ' " " の違いに注目。

⑧ $x と$y の割算。

⑨ $x と$y の整数の割算の余り。%記号

⑩ $x の 2 乗

4.1 初等関数の計算

いくつかの初等関数 (三角関数、平方根 など) は計算できる。 詳しくは、パールのマニュアルの数学関数の項 を見て下さい。　%man perlfunc

結果で 1.2e2 と表示されたらこれは、1.2 x 10**2 という意味だ。

【例題 4.1】　初等関数の計算の例をしめす。(p4-1.pl)

```
#!/usr/bin/perl
print sqrt(4.0),"\n";                          #①

$p = 3.1415926535;    print $p ,"\n";          #②
$p = atan2(1.0,1.0)*4; print $p ,"\n";         #③
print sin($p)," ",cos($p) ,"\n";               #④

print exp(1.0)," ",log(1.0),"\n";              #⑤
print int(1.23)," ",int(-1.23),"\n";           #⑥
print hex(ff),"\n";                            #⑦
print oct(77),"\n";                            #⑧
```

解説

① 平方根の表示
② パイを表現（三角関数は ラヂアンの単位）
③ パイを arctan(1)=π/4 を用いて表現
④ sin(π) cos(π) を表示。tan はない。
⑤ 指数関数、対数関数
　自然対数 e を底とする exp と log
⑥ その他の関数
　整数部のみの表示。
⑦ 16 進数から 10 進数への変換
⑧ 8 進数から 10 進数への変換

4.2 文字変数の計算

文字変数の和は、3.2 で説明した . (ピリオド)が使える。また文字変数の積（整数倍のみ可能、繰り返すのだ）は、x で表す (*ではない)。文字列のなかの一部を取り出す(引き算?) には、 <u>substr( )関数</u>を用いる。

【例題 4.2】 文字変数の計算の例をしめす。(p4-2.pl)

```
#!/usr/bin/perl
$m = 'Arg';                 #①
print $m x 3,"\n";          #②
$m .= ' ';                  #③
print $m x 3,"\n";          #④
$m = 'Lys' . $m;            #⑤
print $m x 3,"\n";          #⑥
$m = substr($m,0,6);        #⑦
print $m x 3,"\n";          #⑧
$m .= ',';                  #⑨
print $m x 3,"\n";          #⑩
@a =split(/,/,$m x 3);      #⑪
print @a,"\n";              #⑫
$a[1] ='IleArg';            #⑬
print "[" . join("---",@a) . "]". "\n";     #⑭
```

① $m に 「Arg」 を代入する。
② $m を 3 回繰り返し印刷する。
③ $m に space を代入する。$m = $m . ' ' と同じ。
④ $m を 3 回繰り返し印刷する。
⑤ $m の前に文字を足す方法を示す。
⑥ $m を 3 回繰り返し印刷する。
⑦ substr($m,0,6) は、$m の 最初の文字 (=0) から 6 バイト文字抜きとる。
(半角英数字は 1 文字＝1 バイト、漢字、平仮名は、1 文字 = 2 バイトだ。)ここでは、「LysArg」が 6 文字 = 6 バイト になる。
　結局この操作では$m から 最後の , （コンマ）を再びとることになる。
⑧ $m を 3 回繰り返し印刷する。
⑨ $m に , を代入する。,を区切り記号に使える。
⑩ $m を 3 回繰り返し印刷する。
⑪ split(3.2 参照)で分割して配列に代入する。
⑫配列では、, がなくなっている。区切り文字に指定したからだ。
⑬ 配列の 2 番目の要素を変える。
⑭join 関数は split の反対だ。join ("aaa", @a) 配列 @a の 要素の間(間だけ)を "aaa"でつなぎなおす。<u>結果は配列でなく変数になる</u>。データの切れ目としての"aaa" に"\t"タブが良く使う。
@a を join でまとめておしまい。

以下に p4-2.pl の実行結果を示す。

```
% p4-2.pl
ArgArgArg
Arg Arg Arg
LysArg LysArg LysArg
LysArgLysArgLysArg
LysArg,LysArg,LysArg,
LysArgLysArgLysArg
[LysArg---IleArg---LysArg]
```

注意:文字列の掛け算に用いる x は必ず前後にスペースをいれること。'a'x3 とかくと、Perl では変数として x3 が使われているのか、文字変数の積 x 3 か区別できないのでエラーとなる。

---

5. 文字変数の計算・応用編・置換と検索
5.0 置換

ある特定の文字を別の文字に変更することは、Perl を使う場合良くある。置き換えは、

　　　　s/パターン/置換したい字/

で行う。なおこの書き方は Unix コマンドの sed の書き方と一緒である。

【例題 5.0】　全角の数字 3 文字表記のアミノ酸、Ile、Leu、Val を一文字の配列 I,L,V に置き換えます。(p5-0.pl)

```
#!/usr/bin/perl
while (<>) {                    #①
s/Ile/I/go;                     #②
s/Leu/L/go;                     #②
s/Val/V/go;                     #②
print;                          #③
}                               #④
```

① 標準入力から 1 行づつ読んで以下を処理
② Ile を I に換える。g オプションは 1 行に 2 個以上あっても処理するというオプション。同様に Leu / Val も以下の行で処理する。
③結果を印刷する。
④ while 。

ここで、g と o は変換の際のオプションで、
g = grobal (1 行の中で該当する全部を変換する。)
o = one time (検索するパターンが、変数でないので Perl に 1 度だけ、コンパイルさせる。処理速度が速くなる。)

となっている。オプションがある場合と無い場合の違いについて、 実験してみると良い。
なおここで、パターン検索の処理はすべて暗黙の配列$_について行われている。別の配列について行いたい場合は

```
$a =~ s/Ile/I/go;
```

のように記述する。

p5-0.dat を以下の用に作り、実行する。

```
% vi p5-0.dat
1  Arg Ile Val Ala Leu His His His Ala Ile
11  Lys Lys His Val Val Leu Leu

% p5-0.pl < p5-0.dat
```

結果はどうなったであろうか？
例えばある文字列を機械的に置換するだけであれば p5-0.pl のように変換規則をただ並べれば良いので、処理時間を特に 気にしなければ簡単にプログラムが作れる。
処理時間を調べたい時には、
% time p5-0.pl < p5-0.dat
でできる。0.020u 0.060s 0:00.09 は それぞれ user の使用した時間、system が利用した時間、その合計だ(単位=秒)。

5.1 パターンの取り出し方

表から特定の列の部分を取り出すのは、2.3 で split を用いて 配列に代入する方法が有効である。ここでは、特定のパターンが含まれる行だけをとりだして、その情報を加工する方法について説明する。結局のところ、Bioinformatics や構造生物学で使うデータファイルは、数千行にも及ぶことが多いため、エディターなどで人力でチェックするのには限りがある。従って、Perl の使い方としてもっとも多いのは、このパターン検索と、次に示す if 構文との組合せ使用である。使い方はスラッシュと括弧()を組み合わせて使う。

【例題 5.1】　特定の部分を $_ から取り出して印刷する。(p5-1.pl)
通常は while(<>) {....} とするが、ここでは $_ の内容がわかり やすいようにプログラムを作った。

```
#!/usr/bin/perl
$_ = '030602 z602 this is a test sentence, a603, yeah !!';
                                                  #①
print /2 (.*)test/,"\n";                          #②
@a = /2 (.*)test/; print @a, "\n";                #③
print /.602(.*)/,"\n";                            #④
print /[a-z]602(.*)/,"\n";                        #⑤
print /\w602(.*)/,"\n";                           #⑥
print /[a-z]60[23](.*)/,"\n";                     #⑦
```

解説
① $_ を定義
②2 と test の間にある部分が抽出される抽出されたところが印刷。
③ ( ) の部分は、配列 @a に入る。
④ . ピリオドは任意の 1 文字で、0602 に続く行の残りを印刷
⑤ 最初の 1 文字は、a-z の文字に限定する。z602 にしかヒットしない。
⑥ これも可で 5 に同じ。\w はアルファベット文字を指す。
⑦ z602 か a603 の後ろのみ印刷。

$_のなかから条件に合致する(...) ではさまれている部分が順に取り出される。取り出された部分は、(...) が現れた順に $1、$2 ... という予約された変数に入る。単に切り出すだけでなくて、置換の中でも使える。s/../../ の中で使用し たい場合には、\1、\2、として指定する。

(例: s/ab(.*)cde/AB\1CDE/ では ab と cde に挟まれた任意の文字を AB と CDE に挟むように置き換えるものである。)
ここで、.（ピリオド）は任意の 1 文字である。* は

0 回以上の繰り返しである。従って、.* は 任意の文字列 となる。+ は 1 回以上の繰り返しである。

[...] は その中から範囲で選択となる。[0-9]+ の様に記すと、数字の繰 り返しになる。これは\d (数字) で置き換えられる。また [a-z] は \w(アルファベット文字 と _) で置き換えられる。

このような 表現は<u>正規表現</u>[regular expression]と呼ばれる。詳しくは、マニュアルを参考にして下さい。正規表現は sed / grep / gawk などにも共通であり、これらを理解すると Linux を用いたデータ処理が飛躍的に向上する。
man perlregep
man perlre

5.2 if 構文での使用法

もしパターンがあったら、処理をするという形は、

```
if(/パターン/){...}
        # パターン が あったら {...} を処理。
if(s/パターン/置換/){...}
        # パターンがあったら置換を行い {...} を処理。
```

という文で行われる。置換をして処理することもできる。【注:if は while や for などと同じ制御文であり、if(--条件式--) {--命令文--} elsif {--命令文--}という文法形式をとる。命令文のところはセミコロンで区切ればいくつ命令を入れても良いが、それ以外にセミコロンを入れてはいけない。/パターン/は、暗黙の変数$_のなかにパターンがあるかどうかを調べる条件式である。s/パターン/置換/は$_に対してまず置換を行い、置換が行われたかどうかを調べる条件式である。】
これは、パターンがあった場合には、/パターン/ が、if の文で 1(真) を与え、 また s/パターン/置換/ によって置換がうまくできた場合には、if の文で 1(真) を与えるからである。だから
print /パターン/;
とすると、ブランクか 1 が出力されるはずである。

【例題 5.2】　$_ のなかに、特定の pattern があった場合に処理をする。また、pattern があった場合があった場合どのような値を返すかを示す。
(p5-2.pl)

```
#!/usr/bin/perl
$_ = 'h9510245 Hiroaki 125 24 35 Keisoku';
                        #⓪
if (/Hiroaki/) { print "Hiroaki san is found.\n" }
                        #①
if (s/Hiroaki/HIROAKI/) { print "Found: ==> ", $_,
"\n" }                            #②
print "Found: ==> ",$_,"\n" if (s/HIROAKI/Hiroak
i/);                              #③
print /Keisoku/,"\n";             #④
print /Keizoku/,"\n";             #⑤
```

解説
⓪ $_ を定義
①もし$_ に Hiroaki があればあったことを印刷。
②もし $_ に Hiroaki があれば Saitou を HIROAKI に 変換して印刷。
③こういう文法も ok だ。この場合でも 置換をさきにする。
④/Keisoku/ がある場合にはどんな値を返すかを示す。
⑤/Keizoku/ はない。この場合にはどんな値を返すかを示す。

実行結果を示す。

```
% p5-2.pl
Hiroaki san is found.
Found: ==> h9510245 HIROAKI 125 24 35
Keisoku
Found: ==> h9510245 Hiroaki 125 24 35 Keisoku
1
0    <== パターンがない場合には、0
%
```

6.　制御文ことはじめ　条件分岐とコマンドラインからのパラメータ受け取り

6.0　if の分岐

条件によって処理をしたり、しなかったりするのには、if が使われることは 5.2 でも学んだ。復習すると、その形式は、

if (--条件式--) { 式が真の場合の処理 }

である。条件としては、
(1) パターンが存在すること、
(2) 2 つの数字、文字の比較または大小関係
が用いられる。(1) や (2) の結果 として真 であれば if の (条件) は 1 偽であれば 0 が与えられる。

【例題 6.0】　if の分岐の例と、条件の例を示す。
(p6-0.pl)

```
#!/usr/bin/perl
$a = 1; print '$a = ',$a,"\n";                #①
if ($a == 1) { print '$a = 1',"\n"; }         #②
if($a) { print '$a is true',"\n"; }           #②
if ( $a != 2 ) { print '$a is not 2.',"\n";}          # ③
if ( $a >= -1) { print '$a is greater or equal to -
1.',"\n";}                                    # ④
$b = 'uec'; print '$b = ',$b,"\n";            #⑤
if ($b eq 'uec' ) { print '$b is equal to \'uec\'.
',"\n"} ;                                     #⑥
if ($b ne 'uec' ) { print '$b is not uec.',"\n"} else
{print 'No',"\n"};                            #⑥
if ("$b" lt 'ued' ) { print '$b is litter than \'ued\' .
"\n"; }              #⑦
if ( length($b) >= 3 ) { print 'The length of $b is >
or = to 3' . "\n";}                           #⑧
$_ = $b;
if ( /ec/ ) { print 'pattern ec is found',"\n"; }
                                              #⑨
if (! /wc/ ) { print 'pattern wc is not found',"\n";}
                                              #⑩
```

解説
① 変数 $a に 1 を代入。
②もし $a が 1 に 等しかったら (== 数値として等しい) 印刷。$a が 1 の判定であるならば、下の表記でも正しい
③等しくないという場合には、!= が使われる。
④大小関係においては、> と < (等号を含まない場合)、<= と >= (含む場合)
⑤次に文字列の条件。数字の場合と違うので注意。
⑤ $b に 文字列 uec を代入。
⑥等しい場合 eq、 等しくない場合 ne
⑦文字列の大小 gt と lt、及び等しい場合を含む ge と le は、アルファベット順で決められる。
⑧ 文字列の長さ (数値) の比較は、長さを与える関数 length() が使われる。
⑨ $_ に代入すれば、5.3 で示した パターンがあるかないかを使える。これは「ec という パターンが $_ にあれば」の例...

特殊な例として、if(1) は常に真であり、if(0)は 常に偽である。 文字の場合には、文字が存在すれば if("a") 常に真、 文字列がない場合 if("") には 常に偽である。 if(<>) は、標準入力から文字があれば正しい。が特殊な場合として数字の 0 という文字列があれば、if("0")が(数値として)偽である点に注意したい。 Perl の場合には、このように 数字と文字を曖昧に扱うことが許されている。

6.1 elsif (注意! elseif でない) 及び else 文

if で正しくない場合には、else 以下を実行する。else は無くても良い。ない場合には自動的に省略とみなされ、次の文の処理に進む。 【構文１】
または elsif で条件をいくらでも分けることができる。【構文２】

【構文１】
if (条件) { 真の場合の処理 }　 else　{ 偽の場合の処理 }

【構文２】
if (条件 1) { 条件 1 = 真の場合の処理 }
elsif (条件 2) { 条件 1 = 偽 で 条件 2 = 真の場合の処理 }
elsif (条件 3) { 条件 1,2 = 偽 で 条件 3 = 真の場合の処理 }
else {全て偽の場合の処理}

つまり elsif は何個あっても良い。場合分けをする場合には、elsif で長い 1 つの if 文を作った方が 、複数の if 文に分けるより処理上速い。
（また 2 つの条件において、and や or の機能を && と | | で、つなげて一つの条件にする事ができる。）

注意: elsif は elseif と書き間違えないように。csh や fortran を知っている人は特に注意。

【例題 6.1】　if – elsif－ else 文の例を示す。画面に何か入力して欲しい。(p6-1.pl)

```
#!/usr/bin/perl
$_ = <>;                         #①
chop;                            #②
if ( /[yY].*/ ) { print 'you said yes',"\n";}          #③
elsif( /[nN].*/ ) { print 'you said NO',"\n";}         #④
else{print "$_", 'please answer yes or no',"\n"}
                                              #⑤
```

解説
① 一行入力し、$_ にいれる。
②$y についている改行の記号を取る。これで $y は文字だけになる。
③入力した文字が y または Y で 始まれば yes と判断する。
④でなければ n または N で始まれば no の判断する。
⑤どちらでもなければ注意する。

なお②で出てきた chop は$_に働く、改行を取る関数である。一行を入力した場合には、その文字変数には、 改行の記号が最後についてしまう。文字が等しいか確かめたいときには、改行の 記号は不要

であるので、chop で落とす。$_ 以外の場合には、関数の普通の 使い方として、chop($a); のように使う。

6.2　コマンドラインからの引数の入力の受け渡し

コマンドで引数(ひきすう) を伴う場合がある。例えば、Unix/LINUX で file を消す場合、

```
% rm file
```
であり、file が引数になる。 また

```
% cp file1 file2
```
のように 2 つの引数 file1 file2 を 持つ場合がある。自分で Perl で小さなプログラムを書いた場合でも、ちょっとした条件などをコマンドラインから入れられたら便利なことが多い。Perl のコマンドでも引数を伴って処理をする場合を考えよう。

【例題 6.2】　引数のパターンがあるものを標準入力から探しその行を印刷する。 (p6-2.pl)

```
#!/usr/bin/perl
$p = shift ;                              #①
while(<>) { print if (/$p/); }            #②
```

①　shift は最初の引数を与える。
②　もし各行に 引数のパターンがあれば印刷

これは Unix/LINUX の grep の コマンドと同じ機能を持っている。（grep の 2 番目の引数は file 名であるが、file からの入出力は、8 章 で示す。） 実行は

```
% p6-2.pl abc < file
```

Perl で引数は 配列 @ARGV で与えられる。つまり、Perl のプログラムを起動したときに、引数があると、それは自動的に名前の予約されている配列変数 @ARGV, $ARGV[0], $ARGV[1] .... にかくのうされる。(csh に似ている)。なおプログラムの名前（自分自身）は $0 に入る。引数の個数は、$#ARGV である。【注：一般に配列 @a の長さは、$#a で 与えられる。】

【例題 6.3】　コマンドの引数を表示する。(p6-3.pl)

```
#!/usr/bin/perl
print "$0 = command name, which has $#ARGV arguments. Arguments are as follows : @ARGV \n";
print "The first argument is $ARGV[0] \n";
```

実際にコマンドを実行してみよ。

```
% p6-3.pl ok a b c
```

7.　その他の制御構造　for、while とループからの中途脱出

7.0　繰り返し for

繰り返しの for の構文は、c 言語の for とほぼ同じ表現である。

for (初期設定; 条件(終値); 変化の方法) { 繰り返す内容 }

の文法である。普通の場合には、初期設定で、初期値を代入(例えば $i=1;)して、 条件では不等式(例えば $i <= n;)、変化の方法では増加 (例えば $i++;) が用いられる。以下に標準的な例を示す。

【例題 7.0】
x = a * sin(nθ), y = b * sin(mθ) の点を θ を 変化させて求める。(リサージュ図形である)。(p7-0.pl)

```
#!/usr/bin/perl
$a = 1.0; $b = 1.0; $n = 3; $m = 4;             #①
$thmax = 3.1415926535 * $n * $m * 2.0;          #②
for ( $th = 0; $th < $thmax ; $th += 0.1)       #③
{
$x = $a * sin( $n * $th ); $y = $b * sin( $m * $th );
                                                #④
printf ("%5.3f %5.3f\n",$x,$y);                 #⑤
}                                               #⑥
```

①　$a $b $n $m の初期値を与える。
②　周期として θ の最大値を決める。θ を 0 から θmax まで、0.1 づつ増やしながら実行する。
③　for の開始
④　$x $y を計算。
⑤　printf は書式付印刷のこと。"..." が書式。%5d は 5 桁の数字という意味。
⑥　for の終了。中括弧は別の行にかくと括弧の対応が見やすい。

printf の 書式は、例えば printf() の使い方を見よ。このプログラムを実行すると、(x, y) の組ができるので、

```
% p7-0.pl > a.dat
```

と data file に取って置いて、gnuplot で、

```
% gnuplot
gnuplot> plot 'a.dat' with lines
```

とする。または xmgr (xvgr, maple, plot) 等の グラフィックプログラムを使って、

```
% xmgr a.dat &
```

とすれば良い。xmgr を使うと、さらにいろいろなことができる。

7.1　文字配列の繰り返し

配列 @a のそれぞれの要素に関して処理をする場合にも、添字(そえじ)の数字 を 0 から 配列要素数 $#a まで変化させることができる。
Perl には、この他にもいろいろな方法で表現できる。
一般に言語というものは、同じことを多くの言葉で説明できればできるほど、より複雑で微妙な表現が可能であり、使いこなすほど明快な表現が可能になるが、初心者にとっては、表現が多いほどわかりにくくなる。表現をまず 1 つ覚え言語を使えることが重要である。そして必要に応じてより目的にあった表現を、人のプログラム(言葉)から学ぶのが良い。

【例題 7.1】列 a の要素を一つづつ印刷するのを、いくつかの表現で表す。 (p7-1.pl)

```
#!/usr/bin/perl
@a = ("csh","perl","fortran","c");                    #⓪

for($i=0; $i<=$#a; $i++) {print $a[$i]," ";}
print " 1.\n";                                         #①

@b = @a;
for($i=0; $i<=$#a; $i++) {print shift(@b)," ";}
print " 2.\n";                                         #②

@b = @a;
while(@b){print shift(@b)," ";}
print " 3.\n";                                         #③

@b = @a; $i = $#b + 1;
while ($i){print shift(@b)," ";}
continue {$i--;}
print " 4.\n";                                         #④

@b = @a; $i = $#b + 1;
while($i--) {print shift(@b)," ";};
print " 5.\n";                                         #⑤

foreach $i (@a) { print $i," ";}
print " 6.\n";                                         #⑥
```

解説
⓪　列 @a に 4 つの言葉を入れる。
①　まずは、for を使った場合。$#a は配列の最後の要素の添字。
② for と shift を使って次の様にもかくことができる。 注意事項として sfift をすると配列の中身が変わってしまうので、事前に @b に copy することが絶対必要。for 文の条件式 で$#a である点に注意なぜなら b は変化する。
③　while(@b) @b を while で使う場合には、要素数が入ることを使うと、3 のようになる。shift(@b) では、$b[0] が配列から飛び出てくると考えれば良い。
④　4 は while の変法 で、 continue を用いて 変化の方法を指定できる。ここでは次の while の実行をする前に、continue の処理をする。
⑤　4. は continue のところが無くても表現できる。()の条件式の中で引き算も行ってしまえばいいのだ。
⑥　@a を壊さずに最も簡単な表現は、foreach を使う方法である。

と、まあこれだけの方法がある。

7.2 繰り返しからの途中脱出

繰り返しをしている途中で、ある条件を満たしたとき、繰り返しの作業を 途中で中止したい場合がある、この場合には last を用いる。
last if　(条件);
があると、条件が真になったところでループの途中でもループから抜ける。
また繰り返す処理を途中で中止して、次の繰り返しの条件を見に行きたい場合には、next を用いる。(BASIC 言語 の next ににている。)
ループが 2 重になっていてどちらの ループの next か判断できない場合には、内側のループ の next と判断される。内側のループから外側のループの next や last をかけたい場合には、ループにラベルをつければ良い。ラベルは コロン：を用いて、

```
loop1: for($i=0;$i<5;$ii++) { $j+=$i; next loop1;}
```

の様にかくことができる。(この場合には、next loop1; は省略できることは言うまでもない。)

【例題 7.2】 次の文が NMR で決定された PDB の file であるかどうかをチェックする。 (p7-2.pl)

```
#!/usr/bin/perl
$nmr = 0;  #
```

```
while(<>) {   #

if ((/NMR/o) | | (/solution structure/o))
{ $nmr = 1; print 'This is NMR structure.',"\n";
last; }                                    #①
}
if (! $nmr) { print 'This is not NMR
structure' ,"\n";}                         #②
```

解説
⓪　はじめに$nmr というフラグを 0 にしておく。各行毎に $_ に入力して、
① もし NMR または”solution structure” があったら NMR で決定された構造と判断し while を中断
② 最後までいった場合には、NMR でないという表示をだす。

実行は、% p7-2.pl < test.pdb でできる。pdb ファイルは各自データベースより取得せよ。
繰り返しで作業を中断できると、無駄な処理が無くなるので良い。 中断の条件を、for の 2 番目のところの条件に使い、条件がなりたたなくなったとき for を終了させることもできる。例えば、
while(<>) {...} は
for (;<>;){...} と 同じである。ここで、for の初期設定と次の処理は不要である。次の処理は、判断をするときに($_=<> が) なされているからである。

Perl に慣れてくると、なるべく簡単な表現で表すことが可能になってくるが、 最初の内からそのような表現に凝る必要はない。正確に動けば、あまり速くなくても、表現方法を問わないのが良い(プログラムのソースの書き方は、常に見やすくありたい)。
言語やスタイルには、時とともに流行があるので、1 つの言語に固執しないほうが良い。プログラム言語の言語としての思想を理解するぐらいで、あとはマニュアル片手にプログラムが出来て、煩雑な手作業による繰り返し作業が短縮できればそれでよいと思う。

---

8. ファイル入出力、open と close、UNIX コマンドとの連携

8.0 file 入出力 < F >

file の入出力は、
1. open 文 で開き、
2. close 文で閉じる間の部分でできる。

通常の file は、<> を用いて標準入出力で行えば良いが、
3. 2 つ以上の file から入力する。
4. 名前がわかっている file から入力する。
5. cp コマンドの様に、file 名を引数にする場合

があるから、それらにも対応する必要がある。

【例題 8.0】　file.input から データを読み込み、file.output に結果を 印刷する例を示す。(p8-0.pl)

```
#!/usr/bin/perl
open(IN,"p8.input");                       #①
open(OUT,"> p8.output");                   #②
while( <IN> ) {                            #③
print OUT $_ ; }                           #④
close(OUT);
close (IN);
```

解説
①　p8.input から読むことを IN と定義する。
②　p8.output に強制的に書くことを OUT と定義する
③　IN で定義してオープンしたデータ$_ があ#る限り、以下を実行。このあたり今までの while 文と全く一緒。
④　OUT に $_ を印刷。$_ は省略できる。
⑤　OUT も IN も必ず閉じること。

IN や OUT は ファイル・ハンドル (車のハンドルと同じ)と呼び、 変数と区別するために 大文字で書く($はつけない)。名称は自由に定めることが できる。
open の””中は、Unix/LINUX で用いられる、リダイレクション (>, <, >>, >!) や パイプ (|) が使える。Unix/LINUX のコマンドも 使える。

8.1 コマンドからの file 名入力の方法

perl を Unix/LINUX の 1 つのコマンドとして使う場合、option (-a, -o file, -help) を選択することでいろいろな機能を選ぶ、といった小技が必要になる。この方法を説明しよう。

【例題 8.1】　p8-1.pl は、cat と同じ機能をするが、以下の option を選べる。

-v で現在のバージョンを示す
-h で使い方を示す
-o file で出力先 file を示す。

```
#!/usr/bin/perl
$VER = "1.001";                            #①
$argv = join(' ',@ARGV);                   #②
while ($ARGV[0] =~ /^-/ )                  #③
        { $_ = shift;                      #④
        if (/^-v$/) {                      #⑤
```

```
		die "version is $VER.\n"; }			#⑥
		elsif(/^-(h|help)$/) {			#⑥
		die "Usage: p8-1.pl file -o file.out\n"; } #⑦
		elsif(/^-o$/) {				#⑧
		$outfile = shift; }			#⑨
		else {					#⑩
print "Usage: p8-1.pl file -o file.out\n";		#⑪
		die "Unrecognised option: $_\n"; }	#⑫
}							#⑬
if ($outfile) { open(OUT,"> $outfile") }		#⑮
else { open(OUT,">- "); } #⑯

foreach $file (@ARGV) {			#⑭
		open(IN,"< $file");	# ⑰
		while(< IN >){print OUT;}#⑱
		close(IN);		#⑲
}				#⑳
close(OUT);		#⑳
```

解説

① プログラムの version を示す変数を定義。
② 念のためコマンド行を $argv に join して保存しておく。
③ 複数あると仮定して、全ての$ARGV について。
④ 一語取り出して、(shift)
⑤ もし -v であれば
⑥ version を表示して中断。die は Perl の終了。
⑥ さもなくば -h であれば、（-help も可）。
⑦ 使い方を示し中断。
⑧ さもなくば -o であれば、
⑨ 次の文字を 出力 file とする。
⑩ それ以外の option は正しくない。
⑪ その場合は使い方を示す。
⑫ 理解できなかった option を示し終了
⑬while の終了。

以下の複数の 入力 file に関して、
⑭ @ARGV には、最後に表示されるべき 入力 file 名がある。ここではそれが、複数あると仮定して、7.1 で使った foreach を使って処理する。
⑮ 出力先の設定をする。もし $outfile があれば、それに出力する。さもなくば 標準出力 STDOUT (予約語) に出力。
⑯ >- は標準出力 を意味する。
それからいよいよ
⑰ 入力先を IN で定義
⑱ IN からの入力をそのまま印刷
⑲ open した IN を、close (忘れずに!)
⑳ foreach を終了。
⑳ OUT を close

以下を実行してみよ。

```
% p8-1.pl -v
% p8-1.pl -h
% p8-1.pl -help
% p8-1.pl -hh
% p8-1.pl p-8.input
% p8-1.pl -o toto p-8.input
```

少し長いプログラムの例題であるが、実用性は高い。while() {print;} は、print；と省略できる。print は変数だけでなく 配列も印刷できるので print；は IN で読む全てのデータがはいる。1 行だけ読み込みたい場合には、{$_ = ; print; } とするか、print scalar; とすれば良い。scalar は 配列を変数に変換するので、最初の 1 行が変数に代入される。

8.2 Unix/LINUX コマンドとの連携。

open 文は、"..." の中に Unix/LINUX のコマンドを入れることができるので、 いろいろ使うことができる。

【例題 8-2】 % cat file | sort -n -r の処理を perl の中で実行する。
(p8-2.pl)

```
#!/usr/bin/perl
$file = shift;				#①
open(IN,"< $file");			#②
$a .= $_ while(< IN >);			#③
close(IN);				#④

open(SORT,"|sort -n -r");		#⑤
print SORT $a;				#⑥
close(SORT);				#⑦
```

解説

① 最初の引数を $file とする。
② $file を open する。
③ sort するために IN を全てをつなげ $a に代入。
④close を忘れずに。
⑤ ファイルハンドル SORT を定義する。
⑥ $a を SORT に 印刷することで sort が実現。
⑦ close を忘れずに。

例えば % p8-2.pl p8-2.dat を実行してみよ。

この例題は、あまりおもしろい例ではない。csh でできることは csh だるのがいちばんである。Perl の中でこのようなことをする場合には、

1. 中でいろいろ処理をしたい場合、
2. ひとつのコマンドにしてしまいたい場合。
3. 2 つの file を同時に読み込んで、比べながら処理をする場合

には有効である。perl のなかで csh を動かすのには、system という関数もある。 perl のなかで csh を動かすか、csh のなかで Perl を動かすかは、その場その場に応じて判断して使い分けて欲しい。

ADVANCED

9. 大きなプログラムのつくりかた。サブルーチン sub, ライブラリ

9.0 別のプログラムの挿入。require

プログラムが大きくなってくると、
(1) 初期設定をする部分、
(2) 同じ処理をする部分、
(3) 条件分岐によって来ない場合が多い部分
等をプログラム本体から分離した方が分かりやすくなる。分離したプログラムを取り込むには、require を使う。この場合、 変数名が同じであれば、そのまま使える。

【例題 9-0】 標準入力を読み込んで、p9-hen.pl に書いている変換規則に従って出力する。(p9-0.pl)

```
#!/usr/bin/perl
$_ .= $a while($a=<>) ;
    # 全ての入力行 $a をまとめて $_ に代入する。
require "p9-hen.pl";
    # $_ を p9-hen.pl に書いてある変換規則で変換し
print;
    # 印刷する。
```

変換規則は必要に応じて、いくらでも変えらるので便利だし、また主プログラムは非常に見やすい形になった。以下に p9-hen.pl の例を示す。

```
s/ka/か/go ;          # ka を か にかえます。
s/ki/き/go ;          # ki を き にかえます。
s/(hu|fu)/ふ/go ;
    # hu か fu であれば ふ に変換する。
s/(zya|ja)/じゃ/go ;
    # zya か ja であればじゃに変換する。
1;          # require で呼ばれた最後に必要。
```

【注意: require で呼ばれた file の 最後には、1; がないとエラーとなる。】

適当な入力ファイルを作成して、
% p9-0.pl < toto で 実行してみよ。

9.1 サブルーチン(副プログラム)

非常に良く使われる処理、他のプログラムでも応用可能なプログラムは、 副プログラムとして分離した方が使いやすい。これを subroutine(サブルーチン) と呼ぶ。例えば、abc というサブルーチンは sub abc {...} という形をしていて、親のプログラムから、&abc($x,$y,...) という形で呼ばれる。$x や $y は サブルーチンに引き渡す変数の配列で、サブルーチン側では、暗黙の配列 @_ に入る約束になっている。

サブルーチンから戻ってくる場合に変化していても、変化しなくても良い。 重要な点は、サブルーチンのなかで処理する場合に使う一時的な変数が、<u>親の変数と同じ名前であってはならない</u>ことである。同じだと、変数の値が 変化してしまう。これを避ける為に、local な変数の宣言をして、たとえ同じ変数名であっても、別の変数として扱うことができる。これによってサブルーチンをブラックボックス的に一つのコマンドとして使うことができる。

【例題 9.1】 サブルーチン a($x,$y) では、($x+$y,$x-$y) をする。また、関数 として &tan を定義する。(p9-1.pl)

```
#!/usr/bin/perl
$x = 1; $y = 2;                # $x, $y を定義、
($X, $Y) = &a($x,$y);
    #サブルーチン a を呼ぶ。
print " x,y  : $x, $y\n";
    # $x, $y を印刷。変化していない。
print " X,Y  : $X, $Y\n";
    # 結果の$X, $Y を印刷。
print ' @_   :',"@_\n";
    # 配列 @_ を印刷。すでになにもない。
@b = &a($y,$x);
    # 今度は $y $x と順番を変えて a を呼び @b に
    代入。
print " main : $x, $y\n";
    # $x, $y を印刷。変化していない。
print " main : @b\n";
    # 結果の配列 @b を印刷。
#
print &tan(3.1415/4.0),"\n";
#
# 関数 tan の例
sub a {              # サブルーチン a の始まり。
local($a, $b) = @_;
    # local な変数 $a $b に変数の値を引き渡す。
local($c, $d);
    # local な変数 $c $d を宣言する。
$c = $a + $b ;
$d = $a - $b ;       # 和と差を取る。
print " sub a: $a, $b, $c, $d\n";
    #サブルーチン a の local な変数を印刷。
($c,$d) ;
```

```
    #結果の 配列の値を返す（サブルーチンの戻り
    値）。
}                         # サブルーチン a の終了。
#
sub tan {
local($th) = @_;
    # local な変数 $th に変数の値を引き渡す
sin($th)/cos($th);
    # sin($th)/cos($th) の値を返す。
}
```

subroutine では、何の値を返さないでも良い。逆に沢山の値を返したい場合には、 配列にすべて詰め込むか、local でない global な変数にサブルーチンの中で 代入すれば良い。サブルーチンの中で、grobal な変数を変化させることは、 あまり奨励しない。それは、サブルーチンをポータブルなプログラムの部品として、使えなくなるからである。

sub a{...} 以下を別の file 保存して、メインのプログラムから require 文で統合しても良い。沢山の便利な サブルーチンの集まりをライブラリーと呼ぶ。ライブラリーの場合には、 require した file の最後に 1; をつけるのを忘れないことが必要である。 1; がないと require が成立したか perl が 判定できないからである。

9.2 ライブラリーの作り方、使い方

9.1 のようにして作られた、サブルーチンは既にたくさん容易されていて、 普通の Unix/LINUX の計算機では、/usr/local/lib/perl にある。ここにある *.pl という file は、perl の中で require{*.pl}; とすれば、使うことができる。 それぞれの サブルーチンの使い方は、そのサブルーチンの先頭に書いてあるので、試しにいろいろ使ってみると良い。perl のプログラムのある、directory のライブラリ も 同じ require で呼び出すことができる。どの directory にある ライブラリー ファイル を検索してくれるかを、

```
#!/usr/bin/perl  -I/usr/local/lib/perl
```

のように指定できる。

【例題 9-2】　　ctime.pl を読んで、時刻を LINUX の time コマンドの様に表示せよ。(p9-2.pl)

```
#!/usr/bin/perl
require("ctime.pl");
    # /usr/local/lib/perl/ctime.pl を 呼ぶ。
print &ctime(time) ;
    # 時刻を表示す。time は、1970/1/1 よりの秒数。
```

ライブラリーのなかには、perl の パッケージとは別に配布されているものもあるので、近くの anonymous ftp サイトで取り寄せると良い。例えば Bioinformatics にかかわりの深いものとして、bioperl がある。

例題

/home/share/perl に VP.seq, OT.seq, GST.seq, aamw.txt のファイルがあるので、それらを利用して以下の例題を練習せよ。

【例題１】
ウシのバソプレシン（VP.seq）とオキシトシン（OT.seq）の配列がそれぞれのファイルに１文字表記で保存してある。それを３文字表記に変換する Perl スクリプトを作成せよ。
プログラムは aahenkan1.pl　という名前で作成して
%　aahenkan1.pl VP.seq [return]
というように、コマンド行から変換するファイル名を読み込み、画面上（標準出力）に３文字表記（１文字空白）で見やすく表記するようにせよ。

【例題２】　aahenkan2.pl
例題１を改良して、配列が大文字であっても小文字であっても正しく処理できるようにせよ。
大文字と小文字が混じっているバソプレシンの配列は（VP2.seq）である。

【例題３】　aahenkan3.pl
例題２を改良して、さらに長い配列を読み込んだ場合には２０アミノ酸で改行するようにせよ。
長いアミノ酸の配列として(GST.seq)がある。

【例題４】　aahenkan4.pl
例題３を改良して、蛋白質の分子量を計算し、一番最後に分子量も表示すようにせよ。なお各アミノ酸の分子量は
　　/home/share/perl/aamw.txt
に表がはいっているので、自分のディレクトリにコピーして使用せよ。

【例題５】　aahenkan5.pl
http://www.genome.ad.jp にアクセスして、公開 DB 上の実際の配列を自分のディレクトリにダウンロードして保存して、そのファイルに対して例題４を動かしてみよ。公開 DB にある蛋白質配列のファイルには、説明行がついていたり配列中に空白が挿入してあったりする。それらがあってもエラーしないようにするにはどうすればよいか？各自工夫して

aahenkan5.pl を作成せよ。すべてを Perl で書くことももちろんできるが、UNIX/LINUX コマンドの
grep / awk / sed
を組み合わせて行うこともできる。

【例題６】　aahenkan6.pl
例題５を改良して、もし上記の蛋白質を大腸菌で発現したときに、次のそれぞれの分子量を表示すようにせよ。大腸菌では開始コドンのメチオニンがフォルミル化されていることに留意せよ。また開始コドンの次のアミノ酸の種類によりメチオニンが切断される場合とされない場合があるので両方を出力させよ。

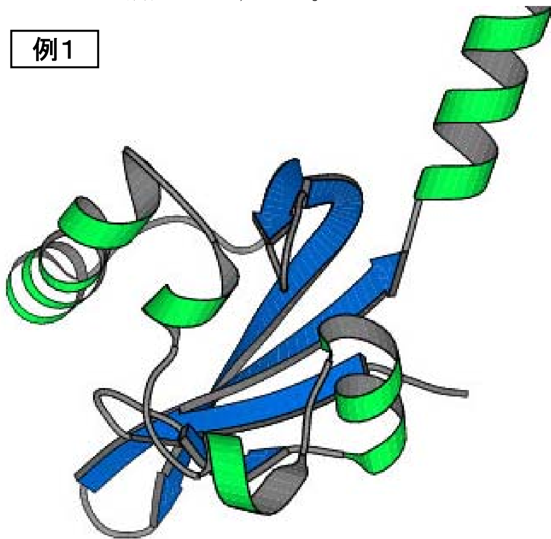
【例題７】　aahenkan7.pl
例題６を改良して、NMR による立体構造解析に用いる安定同位体標識試料の分子量を計算せよ。
以下の分子量を自動的に計算できるようにせよ。
（１）ユニフォーム ^{15}N 標識
（２）ユニフォーム ^{13}C/^{15}N 標識
（３）ユニフォーム ^{13}C/^{15}N/^{2}H 標識
（４）アミノ酸特異的標識で、イソロイシン・ロイシン・バリンのみすべて ^{13}C/^{15}N で標識。
なお標識効率(同位体純度)は標識したものについてはいずれも 100%としてよい。

Linux コマンドから利用するアプリケーション・molscript / gnuplot

廣明　秀一
hiroakih@tsurumi.yokohama-cu.ac.jp
横浜市立大学大学院総合理学研究科生体超分子システム科学専攻
生体超分子計測科学研究室　助教授

§１　MolScript の利用法

MolScript は蛋白質のリボン図を美しく書くためによく使われるソフトであり、RasMol と組み合わせて簡単な図を作成するときに多用される。とくにポストスクリプトプリンタでの印刷に適した見やすいリボン図を作成するのに向いているため、実習のレポート作成や修士論文作成、学会要旨作成、論文投稿などにぜひ活用して欲しい。


例1

ただしこの MolScript というアプリケーションの欠点は、「使いやすい GUI を持っていない」「立体構造 PDB ファイルを読むことはできても、回転することはできない」「できた図を直接画面に表示するわけではない」といった、欠点がある。そのために RasMol や Raster3D(render)といった他のコマンドと併用することになる。

MolScript の操作の流れにそって説明すると

【PDB file】
　↓RasMol
【mol 形式 file】
　↓MolScript
【postscript または raster3D ファイル】

となる。

1. PDB ファイルを用意する
2. RasMol で PDB ファイルを読み込む
   [name@pc150]$　rasmol *hoge.pdb*
3. 分子を回転させて好みの方向にする。
4. molscript ファイルを出力する。
   RasMol> write molscript *hoge.mol*
5. RasMol を終了する。
6. MolScript を走らせる。
   [name@pc150]$　molscript -ps < *hoge.mol* > *hoge.ps*
   [name@pc150]$　molscript < *hoge.mol* > *hoge.ps*
   （ポストスクリプトファイルを出力）
7. できた図を確認する。
   [name@pc150]$　gv *hoge.ps*
   （ポストスクリプトファイルの場合）または
   [name@pc150]$　display *hoge.ps*
8. ポストスクリプトファイルを印刷する。
   [name@pc150]$　lpr *hoge.ps*

なお、RasMol に関しては Windows 版でも機能は同じであるので、Windows 上で上記 1-5 の操作を行ってから Linux 上の MolScript を利用することも可能である。

§２　MolScript ファイルの中身の編集

実は前述のステップ 4 で作成した molscript ファイルを印刷すると、主鎖のリボンだけではなく、側鎖ワイヤー図も表示されてしまって、美しくない（例２）。実際に RasMol が吐いた hoge.mol ファイルの中身を見てみよう。およそ 20 行目以下からファイルのほとんど最後まで

[name@pc150]$　less　*hoge.mol*

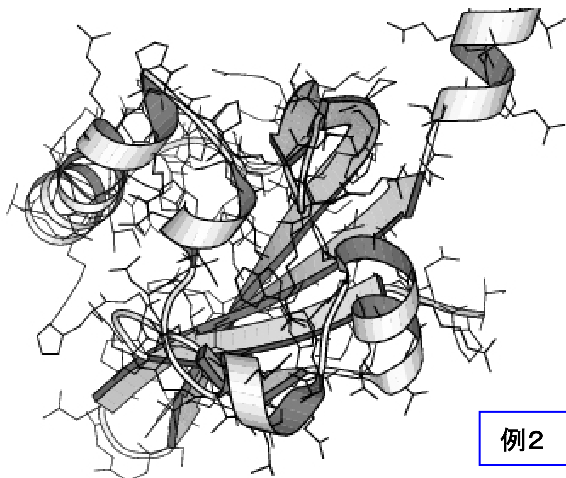
```
 ***************(中略)
bonds require atom O2D and in residues A719
      require atom CGD and in residues A719;

 ***************(中略)
```

という行が並んでいることがわかる。

つまり vi などを用いてこのファイルを編集して、不要な部分を削除することで、シンプルで美しいリボン図を出力させることができるようになる。この場合には一番最後の end plot まで、「bonds」で始まる行と次の行を削除し続ければよい。

【例題１】
100 行以上に及ぶ bonds を含む行を手作業で削除するのは非常に面倒である。いままでに習った linux のコマンドを用いて、これを一瞬で行う方法があるので、それを考えよ。



例2

§３　MolScript ファイルの中身の微調整

MolScript のよいところは、この***.mol ファイルを編集することで、図の色やサイズを変えたりすることができる。この***.mol ファイルの正体は何なのであろうか？それは
[name@pc150]$　molscript [return]
と入力するとわかる。
画面がキーボードからの入力待ちになるのがわかったであろうか？MolScript は対話方式でリボン図を描くときの MolScript 専用のコマンドが入力されるのを待っているのである。先の例で利用した hoge.mol ファイルの中身は、いうなれば MolScript が理解するコマンドを羅列したものである。そこで、エディタで hoge.mol の内容を編集してやることで、画像をさらに好みの形に調製することができる。
[name@pc150]$　vi　*hoge.mol*　としてたとえば以下のような行を追加する。
各文の終わりにセミコロン[;]を忘れないこと！

---

header コマンド (plot 文の直後に)
　background white;　　　(背景を白に)
　background black;　　　(背景を黒に)
　frame off;　　　　　　(frame を消してみる)
位置コマンド (その後に)
　transform atom *
　　　　by centre position res-atom A12 CD1
　　　　by rotation x 180.0　　　　(各軸の回転)
　　　　by rotation y -20.0 ;　　　(rotation 文のおわりにセミコロン)
描画コマンド(ribbon や strand の直前に)
　set linecolour black;　　　(線を黒く)
　set linewidth 3.0;　　　　(線の太さを調整)
　set planecolour orange;（リボンの表面をオレンジ色に)
　set plane2colour rgb 0.0 0.5 1.0;
　　　　(リボンの裏面を RGB の指定色に)

MolScript に理解させるコマンドの書式について、日本語で紹介しているホームページ：
http://o2.biotech.okayama-u.ac.jp/molscript/

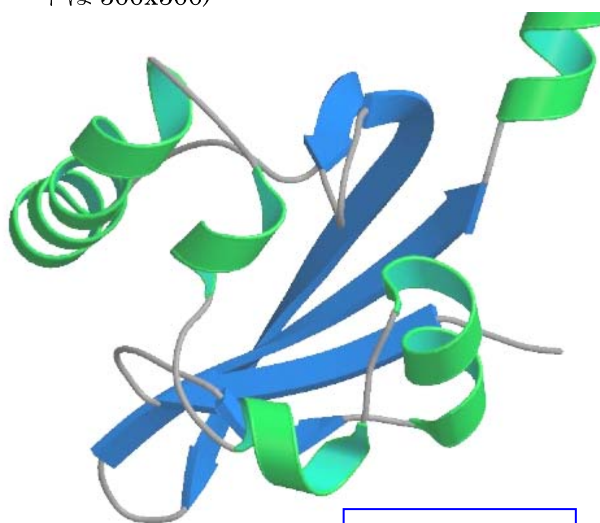
§４　さらに凝った絵を作ってみよう　raster3D を用いてきれいな絵を作る。

raster3D は 3 次元の立体データを、レイ・トレーシング機能によって美しい 2D 画像に描画する(rendering する)ソフトであり、そのコマンドは "render"である。

1. RasMol で構造を読み込む
   rasmol *hoge.pdb*
2. 表示する領域を決める
   RasMol > select all
   RasMol > wireframe off　←一度全部の表示を消す
   RasMol > select within (8.0,Ile15.CD1)
   ←Ile15 の CD1 から 8　以内の原子を選択
   RasMol > wireframe on
   RasMol > centre Ile15.CD1　←Ile15 の CD1 を回転の中心にする
3. 回して気に入った角度にする
4. MolScript ファイルに出力したのち RasMol を終了
   RasMol > write molscript *hoge.mol*
5. MolScript ファイルを編集する
   [name@pc150]$　vi *hoge.mol*
6. MolScript を走らせてその出力を pipe して raster3d でレンダリングする。
   [name@pc150]$　molscript -r < *hoge.mol* | render -tiff *hoge.tif* [return]
   (tiff format で出力)
   [name@pc150]$　molscript -r < *hoge.mol* | render -jpeg *hoge.jpg* [return]
   (jpeg format で出力)
7. できた図を確認する
   [name@pc150]$　display *hoge.tif* [return]
   [name@pc150]$　display *hoge.jpg* [return]
8. 地面をつけてみる
   地面のファイルを作成する
   [name@pc150]$　vi *plane.r3d* [return]
   6 [return]
   50.0 100.0 -20 0.0 -10.0 -80 50.0 -10.0 -80 0.0 0.0 0.0 [return]
   6 [return]
   50.0 -30.0 0.0 -20.0 -10.0 -60.0 100.0 -10.0 -60.0 0.99 0.99 0.7 [return]

9. MolScript を走らせてその出力を raster3d 形式で保存
   [name@pc150]$ molscript -r < *hoge.mol* > *hoge.r3d* [return]
10. 今のファイルを地面と結合して、raster3d に送る。ファイルの結合には cat コマンドを使っているのに注目。
   [name@pc150]$ cat *hoge.r3d plane.r3d* | render -jpeg *hoge2.jpg* [return]
   [name@pc150]$ display *hoge2.jpg* [return]
11. Molscript のコマンドラインオプション
   molscript -○○ < hoge.mol
   のように molscript を呼び出すとき指定するオプションがある。

   -ps　　　ポストスクリプト出力
   -r　　　 raster3D 出力
   -vrml　　VRML2.0 出力
   -size 700 700 700x700 pixel の出力(デフォルトは 500x500)



例3. JPEG の例

詳しい使い方などは
http://www.ntanaka.bio.titech.ac.jp/data.html (東工大。田中研)
などに日本語で使い方が紹介してあるので、参考にするとよい。

【例題２】
適当な蛋白質の PDB から、helix を赤に、strand を青に、coil を灰色にして MolScript でリボン図を作成するためには、hoge.mol ファイルをどのように編集すればよいか、実際にやってみよ。
helix / strand の表と裏の色も変えたい場合は、さらにどうすればよいか？
それらの作業を、いままでに学んだ Linux のコマンドの組み合わせで、簡単に実現する方法を考えよ。

---

§5 gnuplot
数値データを簡単にグラフにする

gnuplot は Linux 上で動作する、グラフ作成ユーティリティーである。テキスト形式で記述してあるデータ(数値の列)を簡単にグラフにすることができるので、他のプログラムで作成したデータを視覚化するときに便利である。また簡単な関数であれば fitting も行える。

1. 起動法
   [name@pc150]$ gnuplot [return]
   コメントが表示されて prompt がでる。
   gnuplot>
2. 終了法
   コマンドプロンプトに対して exit (または quit) と入力。
   gnuplot> exit [return]
3. 数値ファイルをプロットする
   いま data.dat という数値ファイルがあるとする。data.dat の中身は

   ```
   -------data.dat----------
   1.0  12.5
   2.0  13.5
   3.0  10.0
   4.0  10.0
   5.0  17.3
   6.0  4.4
   -----------------------------
   ```
   この内容を plot するには
   gnuplot> plot "data.dat" [return]
4. plot の形式を指定
   gnuplot> plot "data.dat" with line[return]
   gnuplot> plot "data.dat" with spike[return]
5. 関数の plot
   gnuplot では関数を表示できる。
   gnuplot> plot sin(x) [return]
   まず自分で関数を定義してそれを表示することもできる。
   gnuplot> f(x) = sin(x) **2 + 3* sin(x*3) + 5 [return]
   gnuplot> plot f(x) [return]
6. plot 出力を印刷するために postscript を生成。
   gnuplot> set term postscript [return]
   gnuplot> set output "data.ps" [return]
   gnuplot> plot "data.dat" with spike [return]
7. 関数の fitting。
   data file に関数を fit させることができます。
   手順は、(1)関数を定義してやり、(2)それを fit して、(3)それを plot するというふうにやります。
   gnuplot> f(x)=a*x**3 + b*x + c [return]
   gnuplot> fit f(x) "data.dat" via a,b,c [return]
   gnuplot> plot f(x), "data.dat" [return]