第4章　シェルとその機能

第 2 章で述べたように、コマンドインタプリタとはユーザの打ち込む指令に従って各種のプログラムを (プロセスとして) 実行開始させるプログラムである。Unix では伝統的にコマンドインタプリタを「シェル」と呼んでいる。ここではシェルの思想と原理、および各種機能について理解して頂く。シェルを使いこなせるようになると、Unix が自由に操れるようになったという気分になれると思う。

4.1　コマンドインタプリタとその機能

4.1.1　指令入力のユーザインタフェース

前に述べたように、利用者の「次は〜をしたい」という指示を受け取ってそれに対応する動作を起動するプログラムのことを「コマンドインタプリタ」という (図 4.1)。

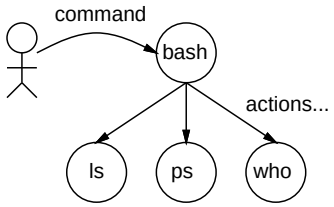

図 4.1: コマンドインタプリタ

ところで、皆様の中にはこれまで Unix を使ってきて「コマンドを覚えるのが大変だ」「コマンドを打ち込むのが負担だ」「だから Unix は使いづらい」と思われている方も多いと思う。しかし、はたしてそうなのだろうか?「使いやすい」というのはどういうことなのだろう?

計算機の世界では、利用者と計算機の間でやりとりをする方式や機構のことを総括的に「ユーザインタフェース」と呼んでいる。PC や Macintosh をご覧になるとわかるように、世の中には様々なユーザインタフェースが存在している。代表的なものを次に示そう。

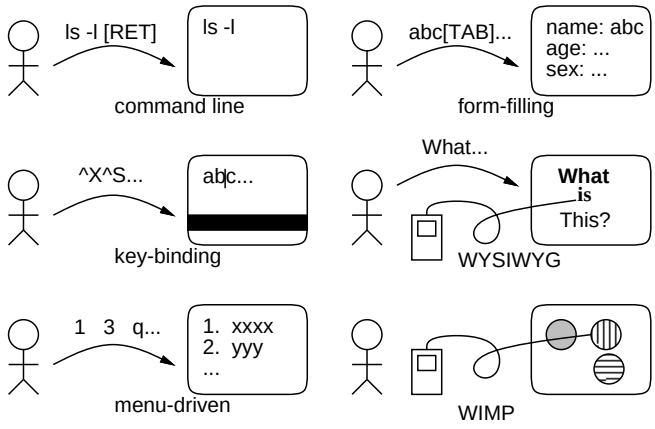

図 4.2: 指令入力の各種方式

- コマンド行方式: Unix のシェルでやっているように、コマンドをキーボードから文字列として打ち込み、最後に [RET] などを押すと実行される。
- キー束縛方式: Mule のように、制御キーを押すと対応する動作が実行されるというもの。テキストエディタなど平文のテキスト入力とコマンドが混在する場合に多く使われる。vi と呼ばれるエディタでは制御キーでなく普通のキーがコマンドに対応している (そのかわりテキスト入力モードと指令モードが分かれている)。
- メニュー方式: メニューが常時画面に表示されているか、または「メニューキー」のようなものを押すと表示され、矢印キーなどで項目を選んで選択すると動作が実行される。
- 書式埋め方式: 画面に「記入欄のある書式」が表示され、矢印キーなどで欄を移動しては欄に適切な文字列を打ち込み、完成したら [RET] キーなどで実行する。座席予約システムなどで多く見られる。
- 直接操作方式: 操作したい対象を画面上で (マウスなどによって) 直接つかんだり移動したりして操作する。ワープロやお絵描きツールなど、「画面で見えるもの」が「最終生成物」に対応させられる場合に使われる。その場合は「What Yous See Is What You Get」(WYSIWYG) と呼ばれる。
- アイコン方式: 操作したい対象を小さな絵など (アイコン) で表し、それをマウスなどで選択し、移動したり重ねたりメニューを出したりして操作する。「Windows, Icons, Menus, Pointing」(WIMP) インタフェースとも呼ばれる。

おそらく、皆様が「使いやすい」と思うのはメニュー方式、直接操作、ア

イコン方式といったあたりだろう。これらはいずれも「マウスなどで対象を直接指示するのでわかりやすい」「メニューで可能な選択肢が向こうから示されるので覚えなくてすむ」という利点を持つので、初めての人にもとりつきやすい。

逆に、Unix で皆様に使っていただいているのはおもにコマンド行方式 (シェル) とキー束縛方式 (mule) で、これらはどちらも何を打ち込むとどんな動作が起こるかを覚えていないと使えない。だから初心者に敷居が高いのは当然である。

ではどうして「使いにくい」コマンド行方式やキー束縛方式が絶滅しないのだろう? それは、初心者には敷居が高い代わりに、習熟するととても高速に操作でき、柔軟性も高いという利点があるからである。これらは逆にいえばメニュー、直接操作、アイコン方式の弱点だということになる。[1] 納得されませんか? 具体的には次の通り。

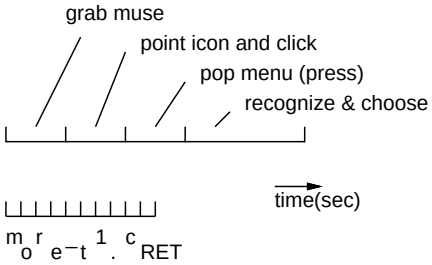

図 4.3: キー入力と WIMP の比較

- 直接操作方式でもアイコン方式でも、操作の対象をマウスで選ぶのには一定 (1 秒前後) の時間が掛かる。また 1 画面に入れておける対象の数は限られていて、画面を切り替える場合にはさらに掛かる。
- メニューから項目を選択するのにも一定の時間が掛かる。このためメニューの一部をキー束縛で選べるように工夫したりする。またアイコン方式でよく使う操作はメニューを経ないで選べるようにしたりする。しかしそうやって用意できる操作の数は限られている。
- メニュー方式では、あらかじめメニューに入れてある動作しか指定できない。また、1 つの (ないし 1 画面ぶんの) メニューに入れられる動作の数には限りがある。アイコン方式でも、1 画面に入れておけるアイコン数は限られている。

[1] 書式埋め方式はこれらの中間で、「どんなものを」打ち込むべきか、またその標準値はいくつか、といった情報は書式によって示せるが、具体的に「何を」打ち込むかは覚えなければならない。

- メニュー方式でもアイコン方式でも動作に対するオプションのようなものは別のやり方で与えなければならない。

これに対し、キーボードの打鍵は 200msec 程度しか時間を要さないので、キー束縛方式では 1 つの動作が高速に指定できる。またコマンド行方式では様々なオプションを自由に指定できる。

もちろん、ここで「だから皆がコマンド行方式やキー束縛方式を使うべきだ」というつもりは毛頭ない。ただ、せっかく Unix を学ぶのだからこれらについて経験を積んでみて、特にその強力さや柔軟性を味わっておいてほしい。そして、お仕事の場では、どんな場合にどんなユーザインタフェースがよいか客観的に判断できるようになって頂きたいということである。

4.1.2　シェルの基本的な機能とプラス α

Unix シェルの基本的な機能は、あくまでも指令を実行させることである。例えば「`ps ax`」といえば、`ps` 指令を `ax` オプションつきで実行させる。それに尽きる。

しかし、それに際して利用者に便利な様々なサービスを追加することで、同じコマンド行インタフェースながらより便利に効率よく作業ができるようになる。具体的には次のような「サービス」がある。

- 複数の指令を組み合わせたり、入出力を切り替えたりする。
- 似たような指令を再度実行するのを容易にする。
- よく使う指令 (群) を簡単に指定できるようにする。
- これらのサービスの様子を個人の好みに合わせて調整する。

そして、利用者にとってはできるだけ単純で覚えることが少ないことが望ましい。それを具体的にどうやっているか以下で見ていくわけだが、その前にシェルの歴史について見ておくことにする。

4.1.3　シェルの歴史

Unix で現在のようなシェルの原型ができたのは 1970 年代初めの Unix version 6 の頃である。versio 6 のシェルはすでに上であげた指令の組み合わせや入出力切り替えの機能を持っていた。

これを受けて、version 7 になったとき AT&T(Unix のもともとの開発元) の Bourne という人が作ったのが Bourne シェルである。Bourne シェルはプログラミング機能が充実しているという特徴を持ち、今でも多く使われている。

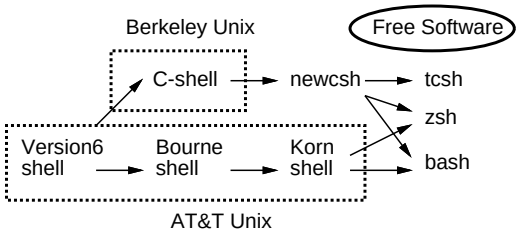

図 4.4: シェルの歴史と系列

一方、カリフォルニア大バークレー校では Unix に様々な改良を施し AT&T とは別にバークレー版 (BSD) として配布した。この中に、version 6 のシェルにコマンド再利用 (履歴) 機能を追加した C シェルが含まれていた。C シェルの履歴機能は評判がよく、広く使われるようになった。ただし C シェルと Bourne シェルは互換性がない。

その後、画面端末が広まった時、シェルでも画面エディタのようにコマンドをキー束縛方式で編集できるようにしたいという要求が出て、多くの人が様々なシェルを開発した。代表的なものには C シェル互換のものとして newcsh、tcsh また Bourne シェル互換のものとして Korn シェル、zsh、bash がある。

Unix にこのように多数のシェルがあるのは、シェルがあくまでも普通のプログラムであり Unix の中核部分とは別に開発でき利用できるからである。これは切磋琢磨という面ではいいのだが、どれを使ったらいいのかという混乱のもとでもある。われわれの所では

```
/bin/sh              --- Bourne シェル
/bin/csh             --- C シェル
/usr/local/bin/tcsh  --- tcsh
/usr/local/bin/bash  --- bash
```

を使っているが、とりあえず普段使っている bash を中心に説明する。ただし bash は Bourne シェルの上位互換なので、対話機能を使わない場合 (シェルスクリプト—後述—を書く場合) にはどちらでも同じである。C シェルとその拡張版である newcsh、tcsh は他のサイトでは広く使われているが、全部やると時間が足りないので割愛させて頂く。基本的な部分は同一である。

4.2 シェルのコマンド実行機能

4.2.1 コマンドとは?

まず最初に、コマンドとは一体何でしょう? `ps` コマンドの表示を見ると `ps` というプログラムがプロセスとして動いているのが見えた。また mule、

kterm、xcolck などもそのまま同名のプログラムだった。つまり、シェルではコマンドとはプログラムの名前に他ならない。より正確に言えば、プログラムの実行形式が格納されているファイルの名前がコマンド名でもある、ということになる。

例えば gcc で C のソースプログラムを翻訳する `a.out` という実行形式ファイルになりましたね? そしてその実行形式を動かすにはファイルの名前 `a.out` をコマンドとして打ち込んでいた。では `a.out` ではなく別の名前だったらどうだろう?

```
% cat t.c
main() { printf("Hello.\n"); }
% gcc t.c
% ls
a.out   t.c
% a.out
Hello.
% mv a.out hello
% ls
hello  t.c
% hello
Hello.
%
```

従って、あなたが新しい指令を作りたければ、そのプログラムの実行形式が入っているファイルの名前をその指令の名前にすればいいわけである。

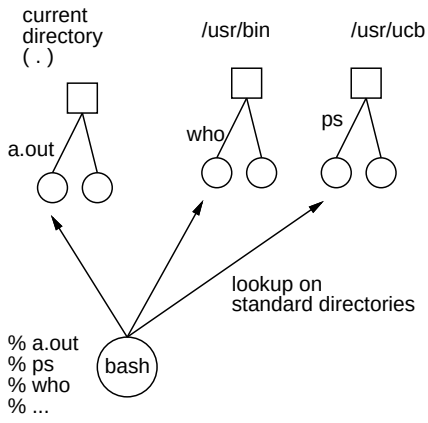

図 4.5: シェルによるコマンド探索

しかしそれにしても、自分は `ls` とか `ps` とかいうファイルは持っていない

が? とおっしゃるかも知れない。もちろん、こういう共通の指令の実行形式ファイルをそれぞれの人が持つのではディスクの無駄だから、共通のコマンドに対応する実行ファイルは共通の場所 (/bin、/usr/bin、/usr/local/bin など) においてあり、シェルはそれらのディレクトリを順番に探して実行形式ファイルを見つけるようになっている。具体的にどこにあるかを知りたければ which という指令を使えば良い。

```
% which ls
/usr/ucb/ls
%
```

それはそうと、自分の作ったプログラムに test という名前をつけないように! なぜなら、test という Unix の標準コマンドと衝突するので。

4.2.2 コマンドの引数とは

さて、コマンドが何であるかはこのようにして分かったが、ではコマンドの引数とは何だろう? 実は、コマンドの引数はコマンドのプログラムに文字列として渡され、あとはそれぞれのプログラムの方で自由に解釈するようになっている。引数を参照する簡単なプログラムを見てみよう。

```
/* t41.c -- print command arguments. */

main(int argc, char *argv[]) {
  int i = 0;
  while(i < argc) {
    printf("%d: %s\n", i, argv[i]); i = i + 1; }
}
```

これまで main には特に引数がないものとしてきたが、実は 2 つの引数を受け取ることでコマンド引数を参照できるようになる。ここで argc は引数の個数を表す整数値、また argv はコマンド引数の並びであり、文字列 (C 言語では文字へのポインタに等しい) の配列となっている。

printf に渡す引数は実は単なる文字列ではなく、その中に「%□」の形のものがあると printf の第 2 引数、第 3 引数、…の値が適切に変換されて埋め込まれ出力される。代表的なものとして次のものがある。

```
%d   --- 整数値を 10 進表記で埋め込む
%x   --- 整数値を 16 進表記で埋め込む
%s   --- 文字列を埋め込む
%f   --- 実数値を小数点形式で埋め込む
%e   --- 実数値を指数形式で埋め込む
```

ともあれ、このプログラムを実行した例を示す。

```
% gcc t61.c
% a.out This is a pen.
0: a.out
1: This
2: is
3: a
4: pen.
%
```

0 番目としてこのプログラム自身の名前も渡されているのがおもしろい。この、コマンド引数を分解して各引数へのポインタを配列に入れるのは、シェルの仕事である (図 4.6)。なお、出力形式は違うが、コマンド `echo` はほとん

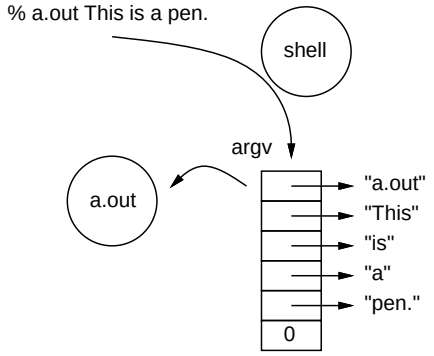

図 4.6: コマンド引数のプログラムへの引き渡し

どこれと同様のことを行うものである。

```
% echo This is a pen.
This is a pen.
%
```

4.2.3　指令の組み合わせとリダイレクト

シェルの特徴の 1 つとして、1 つのコマンド行で複数のプログラムの実行を指示したり、さらにそれらのプログラム群の入出力関係や実行順序を制御できることが挙げられる。

まず、1 つのコマンドに対してその標準入力や標準出力の接続先を切り替えるのには、既に説明したようにリダイレクションを用いる。

コマンド <入力ファイル
コマンド >出力ファイル

両方同時に指定してももちろん構わない。なお、「>」では既に出力ファイルに内容が入っている場合には一担からっぽにされるが、

コマンド >>出力ファイル

とすれば既にある内容の後ろに追加される。
次に、あるコマンドの出力を別のコマンドの入力に接続するにはパイプライン記法による。

コマンド1 | コマンド2 | ... | コマンドn

パイプラインのように入出力の接続をしないで、ただ単に複数のコマンドを実行する場合には; を使用する。

コマンド1 ; コマンド2 ; ... ; コマンドn

; で区切った場合は、前のコマンドが終わってから次のコマンドが実行される。そうではなくて、複数のコマンドを並行して動かしたい場合には&を使う。

コマンド1 & コマンド2 & ... & コマンドn

先に、コマンドの終了を待たずに次のコマンドを入れるには

コマンド &

のように最後に&をつけるというのをやったが、これは上記の特別な場合だったわけである。
そして、ここまでに「コマンド」と書いた部分は再びまた複数のコマンドの組み合わせであってもいい。その組み合わせを制御するには (と) を使う。例えば

% (ps ; ls) >t

とすると、まず ps、続いて ls が実行されるが、これらの出力はまとめてファイル t に書き出されることになる。なお、(と) を使わなかった場合にはまず |が最も強く結びつき、次に ; 最後に&の順番になる。つまり

a | b ; c | d & e | f

は次のものと同等である。

((a | b) ; (c | d)) & (e | f)

4.2.4　ジョブコントロール

前節で述べたように、「&」で区切られた各コマンド (群) は並行して実行されるが、これを Unix では「ジョブ」と呼んでいる。(IBM 文明ではジョブというのは全然別の意味に使われるので注意。) そして、ジョブが作られる時に

```
% mule &
[2] 15449
%
```

のように [1]、[2] などの番号が表示されるが、これがジョブの識別番号である。あるジョブを中止したい時にはいちいちプロセス ID を指定しなくても

```
% kill %1
```

のように%に続けてジョブ番号を指定すればよい。(1 つのジョブが複数のプロセスから成っている場合にもこれでまとめて処理される。なお、kill でシグナルの種類を指定しなかった場合には-TERM を指定したものとして扱われるのに注意。

ところで、Unix では&のあるなしでコマンド (群) の実行終了をシェルが待つかどうかを制御できることは繰り返し説明したが、前者すなわちシェルが実行完了を待っているジョブのことを「フォアグラウンド (前面) ジョブ」、シェルとは並行して動いているジョブのことを「バックグラウンド (裏面) ジョブ」と呼ぶ。そして、それらを互いに入れ換えることも可能である。それには次のようにする。

- バックグラウンド→フォアグラウンド: 「fg %ジョブ番号」を実行する。
- フォアグラウンド→バックグラウンド: まず^Z を打つとフォアグラウンドジョブが kill -STOP を受け取ったのと同様にして凍結される (サスペンド)。ここで「bg」とするとそのジョブはバックグラウンドで解凍されて実行を継続する。

この様子を図 4.7 に示す。

なお、^Z の代わりに^C を打つとフォアグラウンドジョブは kill -TERM を受け取ることになる。だから無限ループなどに陥ったプログラムを中止するには^Z でなく^C を使うこと。(凍結したままだとそのプロセスが使っていた記憶領域は占有されたままなので、これを繰り返しているとどんどん記憶領域が圧迫されてくる。)

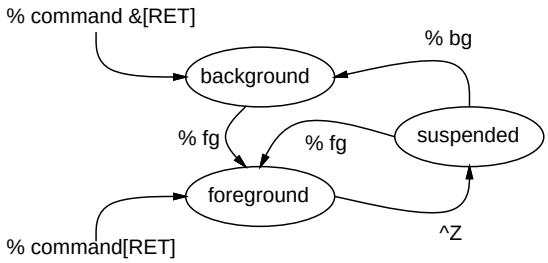

図 4.7: フォアグラウンド実行とバックグラウンド実行

4.3 シェル変数とシェルの調整

4.3.1 シェル変数

多くのプログラミング言語は、値を蓄えておくために「変数」を使用する。シェルは (後で出てくるように) プログラミング言語でもあるので、やはり変数の機能を持っている。変数に値を設定するのには「=」を使用する。

```
% a=0123456789
```

bash など Boune シェル系のシェルでは「=」の前後に空白を入れてはいけないので注意。[2]

一方、変数の内容を参照する場合には変数名の前に「$」をつけることになっている。

```
% cp t.c $a$a$a
% ls
012345678901234567890123456789  t41.c
t.c
%
```

確かに$a のところが変数 a に格納した値 0123456789 で置き代わっていることがわかる。なお、変数の値を調べるにはいちいちファイルを作らなくても引数をそのまま標準出力に打ち出す指令 echo を使えばよい。

```
% echo $a
0123456789
%
```

変数に値を入れて置けるのは分かったが、これは何の役に立つだろう? 例えば:

[2] csh 系では「set a = 0123456789」のように set というコマンドを使う。

- 現在位置付近にないファイルやディレクトリをくり返し参照したいとき、そのパス名を覚えさせておく
- 長いコマンド名や、複雑なコマンド引数などを覚えさせておく

などが考えられる。実はこれに加えて、

- シェルと利用者の間での情報の受け渡しに用いる
- 利用者が動かすプログラムへの情報伝達に用いる

というのがあって、こちらの用途の方が実際にはずっと多い。以下でこれらについて説明しよう。

4.3.2　組み込みシェル変数

シェル変数のうちのいくつかは、利用者が値を設定しなくても最初から値が設定されている。具体的には次のものである。

```
$USER  --- 利用者のユーザ名
$UID   --- 利用者のユーザ ID
$HOME  --- 利用者のホームディレクトリ
$SHELL --- 利用者が使っているシェル
$PATH  --- コマンドを探すディレクトリのリスト
$TERM  --- 端末の種類
$PS1   --- プロンプト文字列
$PS2   --- 〃（継続行用）
```

そして、これらのいくつかは値を設定することによってシェルの動作を好みに合わせて調整できる。たとえば、あなたはプロンプトにカレントディレクトリを表示させたいかも知れない。

```
% PS1='>>> '
>>> PS1='\w> '
~/text/tex/ecs94/sample> PS1='\u@\h '
kuno@smb PS1='% '
%
```

PS1 を設定したとたんに、シェルのプロンプトはその値に変化してしまう。なお、bash ではプロンプト文字列の中にカレントディレクトリなどを表示させるため次のような制御列を書くことができる。

```
\t     現在時刻
\w     カレントディレクトリのパス名
\W     カレントディレクトリ名
```

\u ユーザ名
\h ホスト名

もう 1 つ重要なのは PATH である。

```
% echo $PATH
.:/usr/local/X11R5/bin:/usr/local/bin:/usr/ucb:/usr/bin:/bin
%
```

このように PATH にはコマンドの実行形式が入れてある共通ディレクトリのリストを「:」で区切ってならべたものが入っている。ここに自分のサブディレクトリを追加すると、そのサブディレクトリに入れた実行形式ファイルは他のコマンドと同様に使えるようになる。[3]

```
% PATH=$HOME/bin:$PATH       ← PATH に自分のディレクトリを追加
% mkdir $HOME/bin            ←そのディレクトリを作る
% mv a.out $HOME/bin/hello   ←そこに hello というコマンドを入れる
% hello
Hello.
%
```

これ以外にも、シェルの動作を制御する組み込みシェル変数は多数ある。それらについてはそれぞれの機能の説明のところで述べる。

4.3.3 環境変数

上に述べたように、シェル変数の概念を拡張してシェル自身だけでなくその中から実行したプロセスに情報を渡すのに使用するのにも利用できる。そのようは変数を「環境変数」と呼ぶ。環境変数を使うには、

```
export 変数名
export 変数名=値
```

で変数を環境変数であるとして宣言しないといけない (2 行目の方法では値も一緒に設定できる)。実は上に挙げたシェル変数のうち HOME、USER、PATH、TERM などは環境変数として宣言ずみである。それ以外にも次のようなものが一般に使われる。

```
$EDITOR   --- メールやニュースを打つのに使うエディタ
$PAGER    --- メールなどを表示するのに使うページャ
$PRINTER  --- 標準のプリンタ
$MANPATH  --- man コマンドがマニュアルを検索してくれる場所
```

[3] 実際には複数のマシンタイプでホームディレクトリを共有している場合にはタイプごとに (実行形式ファイルが別個なので) 別のディレクトリを用意する必要がある。「Unix の環境設定」などを見てほしい。

4.3.4　ドットファイル

ここまでに出てきたシェル変数や環境変数などの設定をいちいち打ち込むのは大変である。そこで、ホームディレクトリに「.」で始まるファイル (ドットファイル) を書いておくことでこれらの設定を自動的に行わせることができる。具体的には bash の場合次のファイルがある。

```
$HOME/.bash_profile  --- ログイン時の自動設定
$HOME/.bashrc        --- ログイン時以外の対話モード設定
$HOME/.bash_logout   --- logout 時の作業
```

標準設定では.bash_profile は.bashrc を参照するようにと書いてあるので、当面.bashrc だけ変更して自分の好みの設定をすることができる。標準の.bashrc は次のような内容である。

```
ulimit -c 0     ←コアダンプファイルを作らないように
umask 077       ←標準のファイル保護モードは「go-rwx」
case ...        ←使用するマシンに合わせた設定
  EXPORT PATH=...      ←コマンドパス
  EXPORT MANPATH=...   ←マニュアルページのパス
esac
if [ x$PS1 != x ]; then
  stty erase '^?' kill '^C'
fi
unset MAILCHECK     ←メール到着チェックを OFF に
PS1="\u@\h% "       ←プロンプトの設定
export NNTPSERVER=utogw  ←ニュースサーバ
export EDITOR=mule-nw    ←標準のエディタ
export PAGER=/usr/local/bin/less  ←標準のページャ
export SHELL=/usr/local/bin/bash  ←標準のシェル
export ESHELL=/bin/sh     ← mule のシェル窓用のシェル
export NOMHNPROC=1        ← mh の設定
export METAMAIL_PAGER=/usr/local/bin/less  ← metamail の〃
```

最初の 3 行を除いて大部分がシェル変数/環境変数の設定になっている。この内容は自分の好みに応じて調整してよい。とりあえず、次の 2 つはやっておくようおすすめする。

- PS1 を設定し直して、自分の好みのプロンプトにする。
- PATH に自分のコマンドディレクトリを追加しておくことで自分用のコマンドを自由に作れるようにする。

なお、.bashrc を変更しただけではその変更はすぐには効果を表さない。だから設定内容を吟味するには

```
source $HOME/.bashrc
```

でそれを読み込ませてみる必要がある。

4.4 シェルのユーザサポート機能

4.4.1 ヒストリ機能

ヒストリ機能とは、要するにこれまでに実行したコマンド行を覚えておいて、再度実行するときに最初から打ち込まなくても済むようにしよう、というものである。具体的には

```
% history | less
```

をやってみると、これまでに実行したコマンド群が記憶されていることがわかる。ここで、コマンドの前に書かれている番号を参照して「!番号 [RET]」と打ち込むとそのコマンド行を再度実行させられる。直前のコマンド行であれば「!![RET]」で再実行でき、また「!文字列 [RET]」によれば指定した文字列で始まる最も最近のコマンド行が再実行できる。

ところで、文字列で探した場合には意図していたのと違うコマンド行が見つかってしまうことも多いし、前と同じままではなく少し直したいこともある。その場合には次の方法を使う。

- ^P によって履歴を 1 コマンドずつさかのぼる。行きすぎた時は^N で戻れる。
- または、^R によって探索モードに入り、求めるコマンドの最初の部分を打ち込むと、打ち込んだものと同じ文字列ではじまるコマンド行が連続的に表示される。もし同じ文字列でもっと前のコマンド行が欲しいならそのままの状態で繰り返し^R を打つ。行きすぎたら^S で戻れる。
- いずれの方法ででも、求めるコマンド行が見つかったらそこで次のようなキーを用いてその行を自由に修正できる。

  ```
  ^A    --- カーソルを行頭へ
  ^E    --- カーソルを行末へ
  ^F    --- カーソルを 1 文字右へ
  ^B    --- カーソルを 1 文字左へ
  ^D    --- カーソルのところにある文字を消去
  [DEL] --- カーソルの直前にある文字を消去
  c     --- 文字 c をそのままカーソルの前に挿入
  ```

  修正が完了したら、最後に [RET] でそのコマンド行を実行させられる。

4.4.2　ファイル名展開

これまで、ファイル名の一覧を見るのには ls を使ってきたが、実は次のようにしてもファイル名の一覧が見られる。

```
% echo *
a.out t t.c t41.c
%
```

これはなぜかというと、シェルはコマンド行入力の中に*、?、[...] などの形をした部分があると、これをファイル名をあらわす「ひな型」としてファイル名への展開を行うからである。これらのパターンの意味は次の通り。

```
*       --- 任意の文字列とマッチする
?       --- 任意の 1 文字とマッチする
[...]   --- ... の中のどれか 1 文字とマッチする
```

例えば次の通り。

```
% echo *.c     ←「最後が.c で終るもの」
t.c t41.c
% echo ?.*     ←「2 文字目に . があるもの」
a.out t.c
%
```

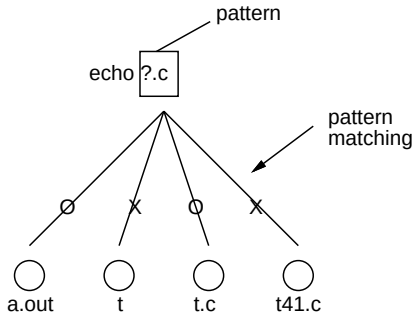

図 4.8: ファイル名のパターンマッチ

なお、注意してほしいのは、ファイル名展開は図 4.8 のようにパターンと存在しているファイル名をつき合わせてマッチするものを取り出すだけであり、従って

```
% cp *.c *.c.bak
```

のようして「新しいファイル名を生成するのにパターンを使う」ことはできないという点である。上のコマンド行がどう展開されるかはその前に echo をつけてみれば分かる:

```
% echo cp *.c *.c.bak
cp t.c t41.c *.c.bak
%
```

これは起こって欲しいことと違うでしょう? 「すべての.c で終わるファイルのバックアップを取る」のには、以下にあるように for を使うのが正しい。

```
% for f in *.c
> do
> echo cp $f $f.bak
> done
```

for 文は in の後に出てきた引数 (これはファイル名展開される) を順番に変数 f に入れながら繰り返し do ... done の間を実行してくれる。(for の説明はすぐ後にも出てくる。)

あと、以下はファイル名展開ではないが bash や csh で使えるパターン「のようなもの」である。

```
~           --- 自分のホームディレクトリに展開される
~ユーザ名   --- そのユーザのホームディレクトリに 〃
{○,…, ○} --- 分配法則 (e.g. a{b,c,d} → ab ac ad となる)
```

4.4.3 脱出文字とクォート

前節で述べたように、*などの文字を含む文字列は展開されてしまうのでそのままではコマンドの引数として渡せない。そこで、渡したい場合には次のどれかを使う。

- 特殊文字の前に\をつける。
- 全体を’または"で囲む。

例えば次の通り。

```
% mv t '[abc]'     ← [や] を含むファイル名
% ls
[abc]   a.out   t.c     t41.c
%
```

この例から分かるように、特殊文字がたくさんある場合にはいちいち\をつけるより引用符で囲む方が楽である。そして、’では変数の展開も起きない ($はそのままになる) が、"で囲んだ場合には変数はその値で置き換えられる。

なお、初心者はシングルクォート (’) とバッククォート (‘) の区別がつかないことがあるが、これは全然違うので注意して欲しい。バッククォートで囲むと、その内部がふつうのコマンドとして実行され、その出力結果で全体が置き換えられる (コマンド置換)。例えば次の通り。

```
% echo "current time is ‘date‘"
current time is Tue May 17 13:20:53 JST 1994
%
```

例から分かるように、"の中ではコマンド置換も起きる (’ の中では起きない)。

4.4.4　コンプリーション

シェルの対話モードで使う強力な機能の仕上げとしてコンプリーション (補完) の説明をしておく。利用者がシェルに向かって打ち込むのはコマンドであり、一方シェルはコマンドとは何と何か知っている。だとしたら、長いコマンド名を全部打ち込まなくても、「わかる所まで打ち込んだらあとはシェルの方で補ってくれる」ことができるわけですよね? この「自動的に補って完成してくれる」ところからこの機能を「補完」と呼ぶ。bash では [TAB] キーを使って補完機能を働かせる。例えば、「mimencode」というやや長いコマンド名を打ち込むとする。

```
% mi[TAB]
% mimencode _
```

つまり、最初の「mi」を打ち込んだ所で [TAB] を押すと残りの部分は bash が補ってくれるわけである。

ところで、こんなことが可能なのは「mi」で始まるコマンドがこれ 1 つしかなかったからで、もし複数あればこうは行かない。その場合には [TAB] の代わりに [ESC]?を打つと「その後に続く候補はこれだけあるよ」という一覧表示をしてくれる。

```
% xl[ESC]?
xload       xlogo       xlsatoms    xlsclients  xlsfonts
% xl_
```

こちらの機能の方がむしろ「最初はこんな感じだったけどあとがわからない」という場合に役に立つ。

さて、コマンド名を入れ終ったらその後はコマンドの引数を (もしあれば) 入れるわけだが、これまで見てきたようにコマンド引数としてファイル名や

ディレクトリ名を打ち込むことが多い。そこで、コマンド引数の位置で [TAB] や [ESC]?を使うと今度はあてはまるファイル名を補ったり表示してくれるようになる。

```
% ls -l [ESC]?
a.out  t      t.c    t41.c
% ls -l t[ESC]?
t      t.c    t41.c
% ls -l t4[TAB]
% ls -l t41.c[RET]
-rw-r--r--  1 kuno          156 May 14 14:25 t41.c
%
```

コンプリーションのありがたい所は、コマンドを途中まで打ったところで「あれ? ファイル名が分らない…」となった時に、打ったコマンドをあきらめて ls を使わなくてもそのままファイルが探せるところである。上の例ではカレントディレクトリのファイルを扱ったが、もっと長い相対パス名や絶対パス名の引数も補完を使いながら完成させて行くことができる。

4.5 シェルスクリプト

4.5.1 スクリプトとは

Unix ではプログラムの標準入力がふだんはキーボードに接続されているが、入力リダイレクトによってこれをファイルに切り替えたりできることをすでに学んだ。ところで、シェルはふだんキーボードからコマンドを読み込んでいるが、これをファイルに切り替えたらどんなことが起こるだろうか?

```
% cat pswho
ps
who
% bash <pswho
  PID TT STAT  TIME COMMAND
29493 p0 IW    0:00 /usr/local/bin/bash
  330 p1 S     0:00 bash
  331 p1 R     0:00 ps
29510 p1 S     0:06 /usr/local/bin/bash
kuno     ttyp0   May 23 11:03   (smri02:0.0)
kuno     ttyp1   May 23 11:04   (smr03:0.0)
%
```

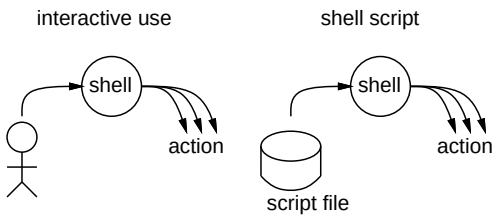

図 4.9: シェルスクリプトの概念

つまりファイルからコマンドが読み込まれて実行されるわけである (図 4.9)。

このように、キーボードから打ち込むかわりに、予めファイルにその内容を用意しておき、それを自動的に順次実行させるようなものをスクリプトと呼ぶ。特に、シェルのためのスクリプトの場合には、シェルスクリプトと呼ばれる。実は、シェルもほかのフィルタと同様、入力ファイルが指定されていたら標準入力の代わりにそのファイルから入力するので、リダイレクトの代わりに単に

```
% bash pswho
```

としても同じである。

さて、シェルスクリプトはいったい何の役に立つだろう? いちいちファイルに打ち込むよりはいきなりキーボードからコマンドを入れる方が簡単ではないか?

- 複雑な指令群を用意する時に、エディタを使って少しずつ試しながら作っていくことができる。
- 何回も繰り返し使う長い指令群をそのつど打ち込まなくても済む。
- 計算機に指令を生成させてそれを実行させられる。

ところで、login/logout 時の設定を行う `.bashrc` などのファイルについて覚えていらっしゃるだろうか? これらもシェルによって自動実行されるので、シェルスクリプトの仲間である。つまり、毎回決まりきった初期設定などを行うためにスクリプトを使用しているわけである。

4.5.2　指令としてのシェルスクリプト

さらに面白いのは、シェルスクリプトを格納したファイルを `chmod` によって「実行可」にしてやるとそのファイルは「シェルによって実行されるプログラム」になる、ということである。たとえば次の通りである。

```
% chmod +x pswho
```

```
% ls -l pswho
-rwxr-xr-x  1 kuno            7 May 23 13:43 pswho
% pswho
  PID TT STAT  TIME COMMAND
29493 p0 IW    0:00 /usr/local/bin/bash
  335 p1 S     0:00 /usr/local/bin/bash
  336 p1 R     0:00 ps
29510 p1 S     0:06 /usr/local/bin/bash
kuno     ttyp0   May 23 11:03   (smri02:0.0)
kuno     ttyp1   May 23 11:04   (smr03:0.0)
%
```

もちろん、このファイルが$PATH に指定されるコマンドディレクトリのどれかに置かれていればどこからでも実行できるようになる。このようにして、いちいち C 言語などでプログラムを書かなくても、自分用の新しいコマンドを増やしていくことができる。

ところで、ここで疑問なのは、Unix にはシェルは多数あったということである。このようにして用意したシェルスクリプトは一体どのシェルによって実行されるのだあろう? 実は特に指定がなければ/bin/sh (Bourne シェル) が使用される。しかし、どれか特定のシェルを使用したい場合には、そのスクリプトの 1 行目に

```
#!シェル名 引数 …
```

という形のものを入れておくことで、実行用シェルを指定できる。これを「インタプリタ指定」と呼ぶ。一般に、実行可能なファイル「f」がテキストファイルでその先頭に「!#コマンド 引数 …」と書かれていた場合、そのファイルをコマンドとして実行すると

```
% コマンド 引数 … [RET]
```

と同じ動作が行われる。「コマンド」は特にシェルのプログラムでなくても何でもよい。たとえば次を見てみよう。

```
% cat abc
#!/bin/ls -lg
% abc
-rwxr-xr-x  1 kuno     faculty        14 May 23 13:47 ./abc
%
```

もっとも実際にはシェルを指定するというのが一番普通ではある。以下ではおもに/bin/sh 用スクリプトを取り上げるが、そのことを明確にするためインタプリタ指定は常に書くことにする。なお、bash と/bin/sh はスクリプト用に使う場合にはほとんど同等である。

4.5.3　スクリプトの引数

ところで、シェルスクリプトによってコマンドが作れるとしても、それに対してさまざまなオプションや引数を渡せなければあまり面白くない。実は、スクリプトをコマンドとして起動したときに、その 1 番目、2 番目、... の引数の値は$1、$2、... というシェル変数に予め設定されている。従って、それを参照したスクリプトを書くことにより引数の値を利用するシェルスクリプトを書くことができる。たとえば、次のシェルスクリプトは指定したファイルについての `ls -lg` を実行するようなものである (引数が指定されなかった場合には、$1、$2、... はすべて空なので通常の `ls -lg` と同じになる。)。

```
% cat lg
#!/bin/sh
ls -lg $1 $2 $3 $4 $5 $6 $7 $8 $9
% lg abc t.c
-rwxr-xr-x  1 kuno     faculty        14 May 23 13:47 abc
-rw-r--r--  1 kuno     faculty        31 May 14 14:16 t.c
%
```

ところで、このままだと引数の数は最大で 9 個までしか参照できない。これでは不便なので、かわりに$*と書くことによって、すべての引数をそこに埋め込むこともできる。

4.5.4　変数の利用

ここまに出てきた例はすべて、単に普通のコマンドを順番に打ち込んで実行させるのとかわりがなかった。しかしシェル変数を一般のプログラミング言語における変数と同様に利用することで色々役に立つ道具が作れる。例えば次は暗算の苦手な人に役に立つ (?) スクリプトの例である。

```
% cat add
#!/bin/sh
sum=‘expr $1 + $2‘
echo "$1 + $2 = $sum"
% add 3 8
3 + 8 = 11
%
```

なお、expr は加減乗除のある式を引数として与えるとその式の値を標準出力に打ち出す。だから単に「`expr 3 + 8`」として使ってもよさそうだが、多数の数の合計のときいちいち+をはさむのが面倒そうである。一方、上の add は引数がいくつでも大丈夫なように改良できる:

```
% cat addn
#!/bin/sh
exp=‘echo $* | sed ’s/ / + /g’‘
sum=‘expr $exp‘
echo "$exp = $sum"
% addn 3 8 12 4
3 + 8 + 12 + 4 = 27
%
```

4.6 シェルスクリプトの制御構造

4.6.1 for 文

ここまで来ると、シェルスクリプトも普通のプログラム言語と同じようにif による場合分けや、while 文などのループ構文があっても不思議はないという気になるであろう。まず最初は前にも出てきた for 文からみてみる。

```
for 変数名 in 値 …
do
  文 …
done
```

for は指定した値の並びから 1 つずつ値を取り出して変数に入れながらループを繰り返す。

これを使って、引数にファイルを指定するとそれらのファイルのバックアップファイルを作るスクリプトを用意してみよう。

```
% cat makebak
#!/bin/sh
for f in $*
do
  cp $f $f.bak
done
kuno% makebak *.c
% ls
makebak         t.c             t41.c
t               t.c.bak         t41.c.bak
%
```

4.6.2　case 文

case 文は、ある値はどのようなパターンにあてはまるかに応じて N 方向に処理を分岐するような構文である。その書き方は次の通り。

```
case 式 in
パターン) 文 …;;
パターン) 文 …;;
...
esac
```

ここで、各パタンの分岐の最後が;; になっているのは、複数のコマンドを順番に実行したいときに 1 個のセミコロンを使用するからである。

これを使って、先の makebak に「-p xxx というオプションが指定されていたらバックアップファイルの末尾を .xxx にする」という機能を追加してみよう。

```
% cat makebak1
#!/bin/sh
post='.bak'
case $1 in
-p) post=$2; shift; shift;;
esac
for f in $*
do
  cp $f $f.$post
done
% makebak1 -p SAVE *.c
% ls
makebak         t               t.c.SAVE        t41.c.SAVE
makebak1        t.c             t41.c
%
```

なお、shift というコマンドは$1 ←$2、$2 ←$3、…のように引数を順繰りに繰り上げる。つまり、第 1 引数が-p だった場合、第 2 引数は末尾の指定なので、これらを捨てて残りをファイル名として扱うわけである。

4.6.3　if 文

case がパターンによる N 方向分岐だったのに対して、if は条件の成否による分岐である。一番簡単な形では

```
if 条件
then
  文 …
fi
```

により、ある条件が成り立った時だけ一連の文を実行する。成り立たなかった場合の動作も指定したい場合には

```
if 条件
then
  文 …
else
  文 …
fi
```

の形になる。さらに、複数の条件により細かく枝分かれする場合も

```
if 条件
then
  文 …
elif 条件
then
  文 …
elif 条件
then
  文 …
else
  文 …
fi
```

のように 1 つの if 文で書くことができる。

ここで「条件」とは何だろう? 普通のプログラミング言語であれば数値の大小比較などが条件として使われるわけだが…

シェルの場合、その特徴は何といっても「プログラムを実行させるための言語」だということである。そこで、while や if の条件も「任意のプログラムを実行させて、その成功/失敗に応じて分岐する」ということになっている。ふだんはほとんど意識されないが、例えば `fgrep` というプログラムは

```
fgrep 文字列 ファイル …
```

のようにして動かすとファイル中の「文字列」を含む行だけを抜きだしてくれるが、この時「1 つでも文字列を含む行が見つかったら成功、そうでなければ失敗」ということになっている。

たとえばこれを使って「ある単語が/usr/dict/words にあるかどうか」調べてくれるスクリプトを書いてみる。

```
% cat word
#!/bin/sh
if fgrep "$1" /usr/dict/words >/dev/null
then
  echo "Sure, I know the word $1."
else
  echo "I don't know the word $1."
fi
% word programing
I don't know the word programing.
% word programming
Sure, I know the word programming.
% word program
Sure, I know the word programming.
%
```

なお、fgrep の出力は不要なので/dev/null という仮想的な出力装置 (ただ出力を食べてしまうだけで何もしない!) に送っている。

4.6.4　test コマンド

このように「一般のプログラムが条件として書ける」というのは面白いけれど、やはり「変数の値が等しいかどうか」など普通のプログラム言語ぽい条件も使いたいと思うことはある。

シェルスクリプトの場合、シェル自身にその種の機能を組み込むのではなく、代わりに文字や数値の比較をやってくれる test というコマンドを利用する。test コマンドはスクリプトで多数使われるため、[という名前も持っている。以下ではこちらの形をおもに使うことにする。その場合、if の条件部は

```
if [ test の扱う条件 ]
then
  ...
```

のような形になる。

具体的に test が扱う条件の形としては、次のようなものがある。

n1 -eq n2　--- 数値 n1 と n2 が等しい (他に-ne、-lt、-le、-gt、-ge)

```
s1 = s2      --- 文字列 s1 と s2 が等しい
s1 != s2     --- 文字列 s1 と s2 が等しくない
-f ファイル名       --- その名前のファイルが存在する
-d ディレクトリ名 --- その名前のディレクトリが存在する
```

これを利用して、先の word のスクリプトを「単語が指定されてなければ利用者に問い合わせる」ように改良してみよう。

```
% cat word1
#!/bin/sh
word=$1
if [ a$word = a ]
then
  echo -n 'Word? '
  read word
fi
if fgrep "$word" /usr/dict/words >/dev/null
then
  echo "Sure, I know the word $word."
else
  echo "I don't know the word $word."
fi
% word1 this
Sure, I know the word this.
% word1
Word? that
Sure, I know the word that.
%
```

すなわち、第 1 引数が空ならば「a$word」は a と同じだから、その時はプロンプトを出して変数 word に語を読み込む (echo の-n オプションはメッセージ打ち出し後改行しないことを意味する。また「read 変数名」は変数に標準入力から値を読み込ませる。

4.6.5 while 文

先の for 文が予め決まった並びに対して実行されるループなのに対し、より一般的なループが while 文である。その形は次のようなものである。

```
while 条件
do
  文 …
```

```
done
```

当然ながら、条件としては一般のプログラムでも、また上述の test でも使える。たとえば、指定した回数だけメッセージを打ち出す例を揚げておく。

```
% cat times
#!/bin/sh
count=$1
message=$2
while [ $count -gt 0 ]
do
  echo $message
  count=‘expr $count - 1‘
done
% times 5 hello
hello
hello
hello
hello
hello
%
```

4.6.6　シェルスクリプトとオプションの扱い

ここで、通常の UNIX コマンドと同じようなオプションを受け付けるシェルスクリプトを書きたいということもよくあるので、そのような枠組みについて簡単に説明しておこう。

実はそれにはさっきやった「$1 がこういう形式だったら$2 をオプションとして取り込む」というのを繰り返し実行させればいい。先のメッセージ打ち出しコマンドを変更して標準のメッセージと回数を用意し、ただし「-c 回数」で回数、「-m メッセージ」でメッセージを指定することもできるようにしたものを示す。

```
% cat times1
#!/bin/sh
count=5
message="Hello."
while true
do
  case $1 in
  -m) message=$2; shift; shift;;
```

```
  -c) count=$2; shift; shift;;
  *) break;;
  esac
done
while [ $count -gt 0 ]
do
  echo $message
  count=‘expr $count - 1‘
done
% times1 -c 3
Hello.
Hello.
Hello.
% times1 -m 'Good-Bye!'
Good-Bye!
Good-Bye!
Good-Bye!
Good-Bye!
Good-Bye!
% times1 -c 2 -m Baka.
Baka.
Baka.
%
```

なお、true というのは「何もしないがいつでも成功して終わる」コマンドで、無限ループを書くのに使う。また break というのはループを終わらせるコマンドである。

4.7 演習

4-1. 自分用のコマンドディレクトリを作成し (「mkdir $HOME/bin」)、.bashrc を変更して、環境変数 PATH に自分用コマンドディレクトリが含まれるようにせよ (case...esac の後に「PATH=$HOME/bin:$PATH」という行を入れる)。コマンドディレクトリに適当な実行ファイルを適当な名前で追加し、その「新しい」コマンドが自分の現在位置に関わらず実行できることを確認せよ。

4-2. 自分用のコマンドディレクトリにシステム標準と同じ名前 (例えば ls など) のコマンドを置いた場合、システムのものとどちらが優先されるか? またはどちらが優先されるかを制御できるか? 実際に試してみよ。

4-3. 1 つのコマンドを何回か実行させた後で bash の漸進サーチ機能をそれらのコマンドを順次、カーソル位置に表示させてみよ。また探している途中で探索文字列を追加したり消したりしてみよ。その様子に基づき、漸進サーチ機能の利点と欠点は何か考察せよ。

4-4. 例題のスクリプト `times1` を改良して、メッセージについては指定がなかった場合利用者に問い合わせてくるように直せ。問い合わせには `read` 機能を使えばよい (例参照)。

4-5. 前問の機能に加え、「`-f` ファイル名」というオプションが指定されていた場合にはそのファイルの中身をメッセージとして打ち出すように直せ。なお、ファイルがちゃんとあるかどうか調べてなければエラーメッセージを打ち出すこと。

4-6. 準備コースでやったように、「`mhmail -s` 主題 宛先 `<`ファイル」というコマンドでファイルの中身をメールとして指定した宛先に送れる。しかしこれだと覚えにくいので、例えば次のような形で使えるシェルスクリプトを用意してみよ。

```
% smail
Address? kuno[RET]
Subject? test[RET]
File? test.txt[RET]
Mail Sent!
%
```

できれば、宛先や主題やファイル名は引数としても指定でき、指定がなかった時のみ問い合わせるほうがかっこいい。なお、テストは自分あてのメールで行い、他人に迷惑を掛けないように。

4-7. `mnews` や `nn` は一旦エディタを起動してしまうので、用意したファイルをそのままニュースに送りたいときには不便である。実は、

```
Newsgroups: グループ名
Distribution:
Subject: 主題ただしローマ字
                                ←空白の行
本文...
...
```

という形のファイルを組み立てて、「`inews -h` ファイル名という形で `inews` コマンドに渡せば指定したグループへのポストが行える。これを利用して前問の smail と同様のインタフェースでニュースへの投稿が行えるコマンドを作ってみよ (正しくないニュースグループを指定したら警告してくれるとなおよい)。

4.8 第 5 回の演習課題その他について

第 5 回の演習課題は次の通りです。

- 課題 4-1〜4-3 のうちから 1 つ。
- 課題 4-4〜4-7 のうちから 1 つ。

来週も月曜日は休みですから、火曜日一杯までに報告を `gssm.k-kiso` に投稿してください。