プログラミング環境 第4回

久野 靖 *

1991.10.22

6 コマンドインタープリタとその機能

6.1 再び、コマンドインタープリタについて

前回までで、計算機環境の基本部分を構成する 2 本柱、つまり:

・プログラムを実行するという機能 -- プロセス。
・情報を恒久的に保持し管理する機能 -- ファイル、ディレクトリ

について一通り見てきた。今回はこれらの「土台」を前提とした上で、もう 1 段上から「自分の意図する仕事を計算機にやらせる」ことについて考えてみる。まず、「計算機と利用者のやりとり」の図を再掲しておく。(Q. ここで、action とは何だったか?また、feedback とは何だったか?)

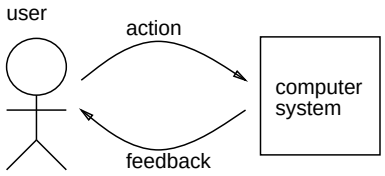

図 1: 計算機と利用者のやりとり

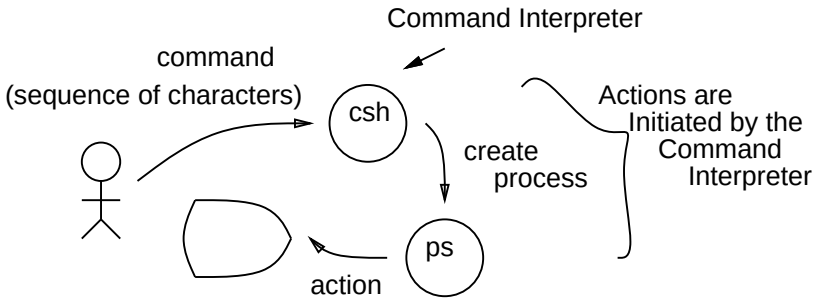

図 2: コマンドインタープリタの概念

さて、例えば座席予約システムとか銀行 ATM など固定した業務の場合にはキー入力も応答も決まったパターンに入るわけだが、我々の場合のように「様々な」仕事を計算機にさせようと思うとそういう固定的なパターンでは済まない。そこで、汎用のシステムとしては、計算機に「こう

*筑波大学経営システム科学専攻

いうことをして欲しい」というメタな情報を文字列として打ち込むと、計算機が「こういうこと」をしてくれるように作るわけである。これをもっと普通にいうと「指令を打ち込む」と「指令が実行される」ということになるわけである。これまた既に述べたように、利用者から指令を表す文字の並びを受けとって、その指令に相応する動作を行なう (より正確に言えば、OS の機能を利用して動作を実現する) プログラムをコマンドインタープリタと呼ぶ。図 2 にコマンドインタープリタの概念図を再掲する。

その特性上、指令というのはいつも決まったパターンだけで済むというものではなく (そういうワンパターンな人もいるだろうけど)、都度様々なことを指定する。だから、それらを柔軟にこなせて、しかも使いやすく分かりやすいようにサポートすることはとても重要である。というわけで、本節では指令を実行する、という「本業」に加えてどんな「$+\alpha$」があり得るかを、Unix の場合を題材に見て欲しい。

6.2 シェル–Unix のコマンドインタープリタ

Unix では伝統的にコマンドインタープリタのことをシェルと呼ぶが、これは「貝殻のように利用者を取り囲み、OS に対する橋渡しをする」ことに由来する... らしい。Unix のシェルにはいくつかの版があって、それぞれに特徴がある。図 3 に主なシェルの系統図を掲げておく。もともと

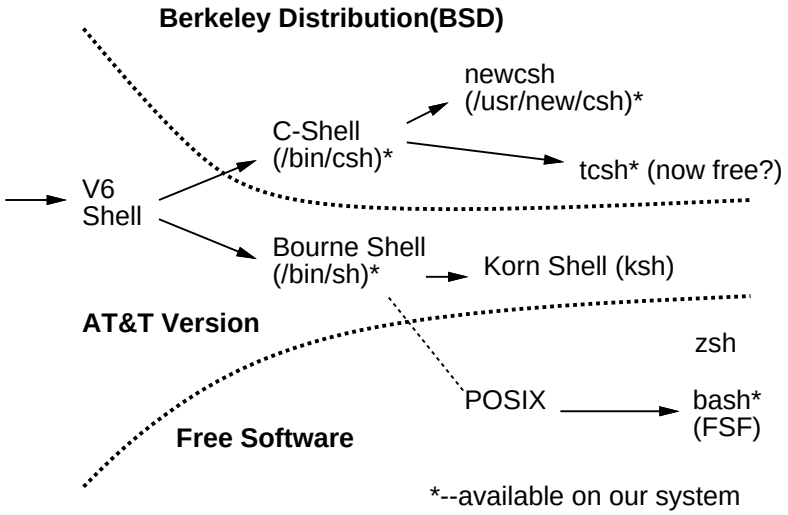

図 3: Unix のシェルの系統図

Unix バージョン 6 あたりまではシェルは一つだったが、その後バークレー版と AT&T 版に Unix が分裂するとともに、バークレー版では C の構文に似せた制御が書ける C-shell が作られ使われるようになった。これはさらに newcsh、tcsh などに発展している。一方 AT&T 版では V6 のものを Bourne が改良したものが作られ、これも標準のシェルとして広く使われている。ところで、これらのシェルはどれも Unix の一部として AT&T やバークレーのライセンスがないと使えないものだったが、最近ではフリーソフト (誰でも自由に使ってよく、ソースからいじれる) のシェルも出てきている。1 つは POSIX という Unix 回りのいろんな事柄を共通規格化しようという団体の案に沿って FSF(Free Software Foundation、nemacs の元となっている GNU emacs を作ったところ) で配布してる bash、プリンストン大の人が作った zsh などもある。我々の所では*印のものが利用できる。これらの特徴を挙げておく:

```
/bin/sh  -- 基本的なシェル。起動が比較的速い。
            スクリプトを書くのに適している。
/bin/csh -- 履歴機能、ジョブ制御 (JCL じゃないよ!) 機能を持つ。
            対話的に利用するのに適している。皆様の標準。
newcsh   -- csh の上位互換。画面上で履歴の表示と指令行の編集が可能。
tcsh     -- newcsh の対抗馬だから、newcsh と機能的には似ている。
bash     -- 画面編集は同様だが、全般には/bin/sh に近い。
```

以下では主として csh/newcsh を対象として説明するが、原理は/bin/sh も同様である。tcsh や bash は興味があれば追求してみていただきたい。

6.3 指令とは?

シェルでは「本業」のコマンド実行は次のような形をしている。

```
指令名  引数  引数  ...  引数
```

散々見てきた `ls -l testd` とか `rm -r abc` とかいうので、最初の `ls` や `rm` が指令名、残りは引数というわけである。引数と引数の間は空白で区切り、この区切りの空白はいくつあってもよい。引数については-で始まるのがオプションで、そうでないのが操作すべきディレクトリやファイル、というパターンが多いが、それは各指令がそれぞれ決めたことで規則として決まっているわけではない。

ところで、それでは指令名、というのは何を意味していると思うか? C や Pascal のプログラムをコンパイルすると `a.out` というファイルができ、それを実行するにはそのファイル名を言う、というのは既に説明した (と思うが?)。次のような具合である。

```
% ls
bin     hello.c work
% cat hello.c
main() { printf("Hello.\n"); }   ←とっても簡単な C プログラム
% cc hello.p                     ← C コンパイラで翻訳
% ls
a.out   bin     hello.c work     ← a.out という実行ファイルができてる
% ls -l a.out
-rwxr-xr-x  1 kuno        24576 Oct 19 17:12 a.out
% a.out
Hello.
%
```

こでこのファイルの名前を変更すると:

```
% mv a.out test1
% test1
Hello.
%
```

これでも同じものが実行できる。つまり、「指令名」というのはファイル名に他ならない。実は ls とか ps とかいうのもすべてファイル名であり、そのファイルに ls や ps の動作を実行するプログラムが入っているだけのこと、というのが真実である。[1] [2]

しかし、自分は a.out というファイルは持っているが ls というファイルは持っていないが?という疑問を持たれることと思う。実は、シェルは「コマンドを探しに行くディレクトリのリスト」を保持していて、その中を順番に探していって「指令名」と同じ名前のファイルが見つかったらそれを実行するようになっている。a.out が実行できるのは、このリストの中に.(現在位置) が含まれているからである。その他、このリストには/bin、/usr/bin、/usr/ucb、/usr/new などが含まれている。例えば `ls` は/bin に入っている。だから、「`ls`」というのと「/bin/ls」というのは全く同じことである。[3]各指令について、それがどこに入っているかを知りたければ

```
which  指令名
```

で知ることができる。

[1] OS によっては指令とファイルは全く別のもので、ファイルに入っているプログラムを実行するには「run ファイル名」のようにそのための指令を使用するものも多い。Unix 方式とどちらがいいと思うか?

[2] ただし、Unix のシェルでもいくつかの指令は別のプログラムではなく、直接シェルの中で実行される。例えば cd などはそうである。なぜか?

[3] 絶対パスで指定する方法は、起動したいプログラムが上記のリストに含まれていない場合でも問題なく使えるという違いはある。

6.4 シェルの指令組み合わせ機能

上で「指令の実行」というシェルの「基本機能」について述べたので、次は「$+\alpha$」の話に移る。最初は、複数の指令を組み合わせる機能である。つまり 1 行に複数の指令が書けたりするわけだが、それをどういう風に書くかについては「規則」、あるいは「言語」が決まっている。その構文を次に示す。[4]

指令 =指令名 [引数]...[< ファイル名][> ファイル名]

指令 =指令 ; 指令

指令 =指令 & 指令

指令 =指令 | 指令

指令 =(指令)

指令 =空

これらの意味は次の通り。まず最初のは「一つの指令を実行する」のを表す。引数はいくつあってもよく、それぞれ互いに空白で区切られる。また、<と>はそれぞれ「入力を端末の代わりにファイルに切替える」、「出力を画面に出す代わりにファイルに保存する」ことを意味する。[5]; はその複数の指令を左から順に実行することを意味する。&は散々出てきたが、その前にある指令の完了を待たないことを意味する。|はその前にある指令の出力を次の指令の入力につなぐことを意味する。最後に、() はかっこである。かっこは何のためにいるかというと、様々な組み合わせ方のくっつき方を制御するためにある。[6](Q.「組み合わせ機能」という「おまけ」は何のために役立つか?)

ところで、シェルが指令を実行するときはそのための子プロセスを作ってその中で指令のプログラムが走ることは既に述べた。また、&つきの場合にはその指令の終了を待たないことも述べた。(図 4、図 5 は前に出てきた図の再掲である。) その他の組み合わせ機能を使用するとそれはどんな風にして実行されるかはなかなか面白い問題である。

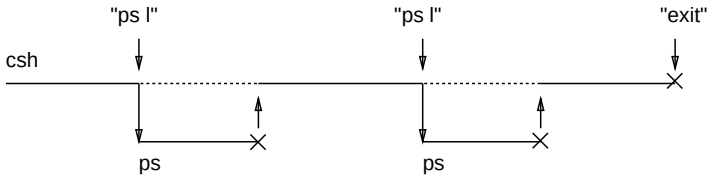

図 4: 通常の指令実行のしくみ

6.5 ジョブコントロール

といっても、「ジョブ制御言語」(JCL) の話では全然ない。&を使って完了を待たない状態で指令を実行できるのは散々やったが、こういう状態で実行している指令 (群) を csh の世界では「バックグラウンドジョブ」と呼び、普通に完了を待ちながら実行しているものを「フォアグラウンドジョブ」と呼ぶ。そして、実は csh はそれらの間を自由に行き来する機能を提供していて、これをジョブコントロールと呼んでいる。ジョブコントロールによって図 6 に示すように、フォアグラウンド/バックグラウンド/凍結状態の間を簡単に行き来できる。ジョブが沢山あるときにはそれを区別するのに「ジョブ番号」を用いて指定できる。これに関連した指令は次の通り:

[4]「計算機言語」というのはいわゆるプログラミング言語に限るわけではない。シェルが受け付ける入力というのも立派な「言語」であり、従って構文規則で表せる。

[5]また、>の代わりに>>とすると既にファイルがある場合にはその末尾に追加される。

[6]たとえば `ps ; who >t` と `(ps; who) >t` はどう違うか考えてみたらすぐ分かる。

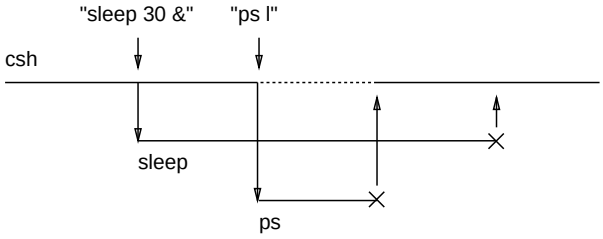

図 5: &つきの指令実行のしくみ

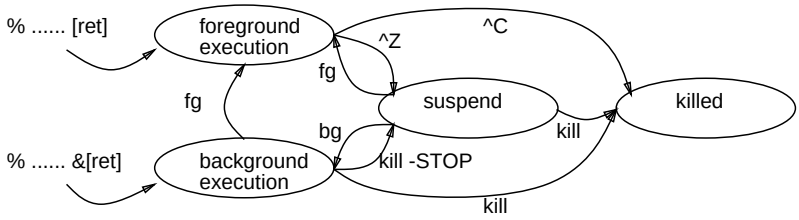

図 6: ジョブコントロールによる状態遷移

```
jobs      -- 現在のジョブの一覧を表示
fg %i     -- ジョブ番号 i をフォアグラウンドで動かす
bg %i     -- ジョブ番号 i をバックグラウンドで動かす
kill [-XXX] %i  -- ジョブに対して kill を実行
```

fg、bg でジョブ番号を指定しなかった場合には「カレントジョブ」(jobs で+符合がついて表示されるもの) が操作される。また kill は以前に説明した kill と同じもので、たたジョブ番号でも指定できるというだけのことである。[7]

6.6 ファイル名マッチング

これまで、ファイル名の一覧を見るには ls を使う、ということにしてきたが、実は ehco *でもファイル名の一覧が見れる。

```
% echo *
bin hello.c test1 work
%
```

では echo というのは何かと言うと、実は単に「引数をそのまま打ち返す」指令である。

```
% echo Hello, World.
Hello, World.
%
```

それではなぜ ehco *でファイル名一覧が見えるかと言うと... 実は、シェルは「引数」の所に次に述べるようなパターン文字があると、それをファイル名とパターンマッチして、見つかったファイル名の群で置き換える、という機能があるためである。だから rm *も「全部のファイルを消す」指令ではなくて、単に*が全部のファイル名に置き換わってから rm に渡るから、rm は「これとこれとこれと... ずいぶん多いなあ」と思いながら (?) 消すだけのことである。パターンには次のようなものがある。

[7] 既に経験された方も多いと思うが、凍結状態のジョブが残ったままで logout しようとするとその旨メッセージが出て警告される。その時は上の指令でこれらを始末するのが「行儀いい」が、再度 logout しようとすればそれらのジョブは消去されて logout できる。

```
*           -- 任意の文字列とマッチする。e.g. h*.p
?           -- 任意の一文字とマッチする。e.g. t??.c
[文字...]   -- 文字のうちどれかとマッチする。e.g.[A-Z]*
```

例えば図7のように rm ab[012].c という指令を打ち込むと、現在位置にあるファイル名のうちでパターン ab[012].c に適合するファイル名のみを選びだし、それでパターンを置き換える。ということは、この場合で言えば rm ab0.c ab1.c と打ち込んだのと全く同じことになるわけである。

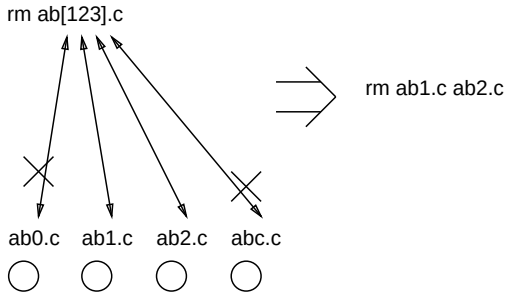

図 7: ファイル名マッチングの原理

これらに加えてパターンではないが、csh では次のような置き換えも行なわれる。

```
~              -- 自分のホームディレクトリ
~ユーザ名      -- そのユーザのホームディレクトリ
{...,...,..} -- 分配法則、e.g. a{b,c} → ab ac
```

6.7 脱出文字、コマンド置換

逆に言えば、上に述べたような特殊文字 (この他もう少しあるが) がファイル名に入っているとそのままではそのファイル名を指定できないが、そのような場合には

・'?*~'、"?*~"のように引用符で囲む。
・\?のように脱出文字「\」を前置する。

のどちらかで特殊文字の機能を殺して「そのまま」に扱うことができる[8]。このことを無視して「?はファイル名にできない」などと思う奴はアホである。

ところで、「'」(シングルクォート) と「`」(バッククォート) の違いに注意。実は「`....`」というのは全く違った意味で、.... の所を指令として実行し、その結果をその場所に埋め込む (コマンド置換) 機能を意味している。これは次に説明するシェル変数にファイルから値を持ってきたりしたい場合に便利な機構である。

6.8 シェル変数

指令を打ち込んでいて、何回も同じ文字列を打ち込むのが煩わしいと感じることがある。そういう時、その文字列を「変数」に入れておいて、あとはその変数を参照することができると便利である。ので、そういう機能がついている:

```
set 変数名 = 値              -- 任意の文字列値を変数にセットする。
set 変数名 = (値 値 ...)     -- 同様だが、値の配列になる。
$変数名                      -- 変数の値を参照する。
$変数名 [数値]               -- 同様だが、配列の n 番目を参照。
```

[8]厳密には"..."はちょっと特別だが、それはすぐ後で出てくる。

これが「ファイル名の最大は何文字か」などに便利なのはご存じの通り (Q. 他の使い道を思いつきますか?)。ところで、従って$も特殊文字なので普通の字として使う場合は上の方法で脱出する。がしかし、"..."だけは特別で、この中に現れる変数参照は置換えが起こる (その方が便利だから)。

ところで、一群のファイルに対して指令の列を実行したい、という場合にこれと関連して便利な機能がある。

```
foreach 変数名 ( .... )
指令...
指令...
...
end
```

これは最初のかっこ内の部分にある要素 (これもパターンを使って生成するのが普通) それぞれについて、それを「変数名」の変数に入れては指令群を実行することを繰り返す、というものである。

6.9 特別なシェル変数

シェル変数の中に、ただ「値を覚えておく」だけではなくシェルにとって特別な意味を持つものがいくつか存在する。(なぜ、そういう風になっていると思うか?)

```
$cwd    -- 現在位置が入っている。
$user   -- あなたのユーザ名が入っている。
$home   -- あなたのホームディレクトリが入っている。
$path   -- 指令を探しに行くディレクトリの配列。
$term   -- 端末の種類が入っている。
$prompt -- コマンドプロンプトが入っている。
```

これらの値を変更するには (もちろん普通の変数名だから)set を使う。また値を見るには echo を使えば良い (お分かりかな?)。

6.10 環境変数

上記の特別なシェル変数は、シェルの振舞いを自分の好みに応じて微調整するのに好都合である。同様に他のプログラム (指令) の振舞いの微調整もできると嬉しい。この目的で「環境変数」というのが作られた。環境変数の値の設定は

```
setenv  環境変数名  値
```

で行なう。環境変数は「グローバルなシェル変数」という位置づけなので、その値は他のシェル変数と同様「$変数名」で参照できる。

6.11 alias(別名、または短縮名) 機能

良く使う指令 (または指令の組み合わせ) が長いので打ち込んでいてうんざりすることがある。そういう場合、alias 機能で

```
alias  名前  文字列
```

というのを実行しておくと、「名前」が指令名の部分に現れた時それが機械的に「文字列」に置き換わるようにできる。

6.12 自動実行ファイル

ここまでに出てきたように、シェルにはいろいろ「予めやっておくと便利に使える」機能があるが、これを毎回打ち込むのでは煩わしい。そこで、各自のホームディレクトリに次のようなファイルを用意しておいて自動実行させることにより各種の設定が自動的にできるようになっている。

```
.login      -- login した時に実行される。
.cshrc      -- csh/newcsh がスタートする際に実行される。
.logout     -- logout 時に実行される。
```

これらのファイルは . で始まるので ls -a でないと表示されないことの注意。普通は set/setenv/alias などの設定は.cshrc に記す。ところで、より便利にしようとして.cshrc を変更した場合、その設定は次に起動された際に実行されるわけだが、そのために一旦 logout するのは結構面倒である。そこで次の指令を使うと便利である。

```
source  ファイル名    -- ファイルから指令を読み実行する
rehash                -- $path の変更に対処する
```

つまり、.cshrc を source して rehash すれば一旦 logout しなくても同様の効果がある。また手で$path の内容を変更した場合にも rehash を実行しないと変化が反映されないので注意。

6.13 csh とヒストリ機能

上に述べてきたような様々な機能を使う場合、利用者が打ち込む指令行はずいぶん長いものになる場合がある。それを複数回やる場合、そのつど打ち込む代わりに前のを再利用したいと思うのが自然である。そこで、.cshrc に

```
set history = 100
set prompt = '... \! ...'
```

のような行を入れておくと、打ち込まれた指令行は記憶されるようになり、かつ入力時にこの指令は「何番目か」を表示してくれるようになる。また、記憶されている指令行は

```
history
```

で表示して見られる。ここで記憶している行を再実行するには

```
!!          -- 直前の指令行を再実行
!番号       -- 指定番目の指令行を再実行
!....       -- 文字列.... で始まる一番最近の指令行を再実行
```

などの指定方法が使える。また、そのまま再実行するのではなくちょっとだけ変更したい場合もあると思うが、その場合には

```
再実行指定:s/xxx/yyy/
```

(ただし「再実行指定」は上の 3 つのどれか) により、指定した行の中に現れる xxx を yyy に置き換えてから再実行できる。この省略形として次のものもある。

```
^xxx^yyy^       -- !!:s/xxx/yyy/ と同じ
```

6.14 newcsh の機能

とはいえ、csh で再実行を指定するのは何となく間接的でもどかしい。そこで csh を拡張して入力行や再実行行の修正と確認が画面端末で便利に使えるようにしたものが newcsh である。情報科学科の皆様は標準が newcsh になっているはずだが、いずれにしても自分が使っているシェルを

```
chsh 自分のユーザ名 新しいシェルのパス名
```

により変更することができる。newcsh でさらに画面編集を使うには

```
set editmode = emacs
```

なる行を.cshrcに入れておく。これで再loginすれば準備完了。上記のように指定した場合、newcshではemacsと同様のコントロールキーで次のような制御ができるようになる。

```
^P/^N    -- これまでに実行した指令の履歴を遡る/戻る
^F/^B    -- 行内でカーソルを右/左に移動
^D/[DEL] -- 1文字消去
普通の字 -- カーソルの場所にその字を挿入
[RET]    -- 現在表示されている行を指令行として実行
```

また、ファイル名が長くて打ち込むのが大変な場合、前述のファイル名マッチングを使うこともできるが、思わぬマッチングが置きて意図しないファイルを選んでしまうことがある。そこで代わりに画面上で確認しながら操作できる、ファイル名補完機能が用意されている。この機能は

```
% rm test_
```

のようにファイル名を途中まで打ち込んだ状態から利用する。ここで現在位置にあるファイルがtest1、test2、testdataの3つだったとしよう。まず、「[ESC]l」により「test」で始まるファイル名の一覧が表示され、入力中の指令行が再表示される:

```
test1  test2  testdata
% rm test_
```

次に消したいのがtestdataであれば、「d」まで打ってから「[ESC][ESC]」により、残りの部分を自動的に埋めて

```
% rm testdata_
```

の状態にしてくれる。あとはこのまま[RET]で実行すればよい(もし意図しないファイル名が見つかったのならやめればよい)。

6.15 シェルの各種機能のまとめ

最初に述べたように、シェルにはコマンドインタープリタとしての基本機能に加えて多くの「$+\alpha$」が備わっていたわけである。それがあまりに多くてごちゃごちゃした印象を与えたかも知れないが、個々の機能についてみればそれぞれ存在意義はちゃんとあると思う。それがばらばらに存在するところは「欲しいと思った人がその機能を追加して組み込む」というUnix発展の歴史的経過から見てやむを得ない点でもあるが、これらを統合して、「小数の機能で同等の柔軟さと使いやすさを提供する」ことを試みるのは興味深いチャレンジになるとは思う。

A 演習、課題

A.1 6節の演習

6-1. /usr/new/csh、/bin/csh、/bin/shなどをコマンドとして実行するとどうなるか。psなどを用いて調べよ(exitもしてみること)。また、これらのシェルの違いを探求してみよ。

6-2. 何か適当なCやPascalのプログラムをコンパイルしてa.outを作り、これを「ls」という名前にしてみよ。「ls」というとどちらが実行されると思うか予想し、実地に試せ。また、同じ名前にしてしまっても区別して起動するにはどうしたらいいか。

6-3. 指令の組み合わせの際に()が役に立つ場合の例を考え、実際に試してみよ。[9]

6-4. 指令の各種組み合わせを行なった場合、それらが実行中のプロセスの親子関係はどういう風になっているか観察し、これらの機能がどうやって実現されているか推理せよ(これはなかなか高度ですよ)。

[9] ヒント:verb|cd XXX; ls と (cd XXX; ls) はどう違うと思うか?

6-5. 自分の適当なディレクトリで、echo を使って次のようなファイルの名前一覧を表示させてみよ:

- C のソースファイル。
- 名前の中に数字を含むようなファイル。
- ちょうど名前の長さが n 文字のファイル。
- 大文字で始まるファイル。

6-6. 変数 path に入っているディレクトリの順序を変更したり数を減らしたりして起きることを体験せよ。とくに `set path=(); rehash` でどのくらい耐えられるかやると面白い。

A.2 4 回目の課題

本日の課題から報告を提出する場合は、上記 6-1〜6-6 のうちから最低 2 つ以上選択して下さい。