# 地球惑星物理学科　計算機演習

～ Unix2 ～

森 晶輝[*1*2]

令和 4 年 4 月 7 日

[*1] 宇宙惑星講座 杉田研究室 修士 2 年

# はじめに

計算機演習も 3 回目となりました。基本的な部分は前回習ったと思いますので、今回は少し発展的だが知っておくと非常に便利な事柄について学んでいきます。また、慣れないことが続き消化不良に陥っている人も居るかと思いますので、コマンドラインでのファイル操作の復習も兼ねて行います。

Unix 端末は、最初はとっつきにくい部分もありますが、使いこなせるようになることで、他の汎用 OS 以上の機能やカスタマイズ性を実現することができます。そのためにシステムの理解は不可欠であり、講義の後半ではそのための基礎的な知識を説明したいと思います。

本テキストは Python の導入にともなって削除された中盤の TA 担当授業 Shell 1[*1], Shell 2[*2]の内容を元の UNIX 2[*3] のテキストにマージし編集して作られています。極めて内容過多かつ先取りをしているため、今日すぐにすべてをマスターする必要はありません。今日は 2 章までをしっかりと身につけていただけることを目標にしています。3 章以降はこれから様々な課題をこなす中でゆっくりと咀嚼していただければと思います。また、分からない内容があったら気軽に TA に聞いてください。

## 配付資料の説明

今日使うファイルは~mori2021/TA2021/exercise/, hosotani2021/, dover:~ohtake2021/shell2/ にあります。とりあえずこのディレクトリをそのまま自分のディレクトリにコピーしましょう[*4]。

```
~$ scp -r dover:/home2/ohtake2021/shell2/ .
```

コピーしたら、以下のコンテンツが揃っているか確認してください。

- words.txt, stations.txt, sample01.sh ～ sample09.sh ::サンプルファイルです。テキストを読みながら適宜使ってください。(sample04.sh と sample08.sh を作るのは練習問題なので、ありません)。
- shell2_2021.pdf ::2021 年版 Shell 2 テキストの電子版です。コピペしたいとかあれば使ってください。
- (ディレクトリ) 2016_02, 2017_02, 2017_11, 2018_01, (ファイル) authors.txt ::提出してもらう課題に関するファイルです。最終章の課題に関する説明を読んでから使ってください。

---

[*1] 2007 年度シェルその 2（吉武さん製作）およびシェルその 3（落さん製作）を参考に作成された 2009 年度テキスト（竹尾さん製作）を参考に作成された 2010 年度テキスト（若松さん製作）を参考にして作成された 2011 年度テキスト（平林さん製作）を参考にした 2013 年度テキスト（末松さん制作）を参考にした 2014 年テキスト (菅野さん作成) を参考にした 2016 年テキスト (松岸さん作成) を参考にした 2018 年テキスト（柳町さん作成）を参考にした 2019 年テキスト (池端さん作成) を参考にした 2020 年テキスト (加藤さん作成) を参考した 2021 年テキスト (細谷さん作成) にしています。

[*2] 2010 年の松井さん、2012 年の小池さん、2013 年の西川さん、2017 年の小澤さん、2018 年の灘本さんが作成してこられた資料の内容に、2020 年と 2021 年に大竹と茂木が加筆修正し、課題を一部変更しました。

[*3] 2008 年に三木順哉さんによって作成されたものを元に作成

[*4] この資料において、ssh コマンドと scp コマンドは 559 室で実施する場合を想定しています。もしリモートで、かつ~/.ssh/config を書いていない場合には、dover の部分を s2226**@dover.eps.s.u-tokyo.ac.jp に書き換えてください。

# 目次

# 第 1 章

# これまでの復習

## 1.1　基本的なコマンドとオプション

ターミナル上で次のようなコマンドを利用して、ファイル操作に慣れましょう。

表 1.1　基本的なコマンド

| コマンド名 | 語源 | 概要 | 重要オプション |
|---|---|---|---|
| cd | Change Directory | カレントディレクトリの移動 | - |
| pwd | Print Working Directory | カレントディレクトリの表示 | |
| ls | LiSt | ファイル一覧の表示 | -l -a -d -h -F |
| cp | CoPy | ファイルの複製 | -R -f |
| mkdir | Make Directory | ディレクトリの作成 | -p |
| mv | MoVe | ファイルの移動 / リネーム | |
| rm | ReMove | ファイルの削除 | -r -f -i |
| rmdir | ReMove DIRectory | ディレクトリの削除 | -p |

前回の講義でオプションについても習いました。例えば、ターミナルで`ls -R`と入力してみましょう。どうなりましたか？ これはどんなオプションでしょうか。`man ls`として、確認してみましょう。また、一度に複数のオプションも使用できましたね。`ls -la`と入力してみてください。これは`ls -a -l` と同じ結果となるはずです。コマンドの機能を忘れてしまったときは`man <コマンド名>`とするか、それでもわからなければ`chrome &`などでブラウザを呼び出して Google などで検索してみましょう。

mac上で safariを使えばよい

```
合計 108
drwxr-xr-x  12 ueda student  4096  4 月 10 14:24 ./
drwxr-xr-x 305 root root    12288  3 月 31 17:59 ../
-rw-------   1 ueda ta         51  4 月 10 14:24 .Xauthority
-rw-r--r--   1 ueda ta       5837  3 月 31 17:47 .Xresources
-rw-r--r--   1 ueda ta        543  3 月 31 17:47 .aliases
drwx------   2 ueda ta       4096  4 月  5 18:20 .anthy/
-rw-------   1 ueda ta       1571  4 月  5 18:49 .bash_history
-rw-r--r--   1 ueda ta        416  3 月 31 17:47 .bash_profile
```

```
-rw-r--r--   1 ueda ta        973  3月 31 17:47 .bashrc
drwx------   3 ueda ta       4096  4月  5 16:07 .dbus/
-rw-r--r--   1 ueda ta       4120  3月 31 17:47 .emacs
drwx------   3 ueda ta       4096  4月  5 17:49 .emacs.d/
drwxr-xr-x   5 ueda ta       4096  3月 31 17:47 .fluxbox/
drwx------   2 ueda ta       4096  4月 10 14:24 .gconf/
drwxr-xr-x   3 ueda ta       4096  4月  5 16:07 .local/
drwxr-xr-x   3 ueda ta       4096  4月  5 18:40 .texmf-var/
drwx------   3 ueda ta       4096  3月 31 17:47 .uim.d/
-rw-r--r--   1 ueda ta       5230  3月 31 17:47 .vimrc
-rw-------   1 ueda ta        577  4月  5 18:47 .xdvirc
```

ここに載せている以外にも様々な隠しファイル (ピリオドで始まるファイル) や前回までに作成したファイルが見えると思います。`ls -l` の出力に関しては前回 (UNIX その 1) の講義で習いましたが、ちゃんと意味が理解できるでしょうか。一列目にある文字はパーミッションを表していますが、その意味を忘れてしまっていたら前の資料を確認しておいてください。また、隠しファイル (.bashrc など) は主にユーザ毎の設定ファイルということでした。隠しファイルの設定に関しては、後ほどもう少し詳しく説明したいと思います。

### 1.1.1 練習

いくつかのコマンドとオプションを使って、遊んでみましょう。例えば、`sl`としてみましょう。これは何のためのコマンドでしょうか？ このコマンドはインストールされていますか？ インストールされていない場合はパッケージマネージャを使ってインストールしてみてください。そして、`man sl`としましょう。また、そこで見つけたオプションを試してみましょう。`cal -jy`とかもやってみましょう。

また`du`は便利なコマンドの 1 つです。`du -sh`として、それぞれのオプションの意味を`man du`で確認してみてください。また、`du -sh *`と入力してみましょう。

(時間が余った人は `finger`や`last`, `who` なども調べてみると面白いかもしれません)

また、ファイル操作に関するコマンドは非常に重要です。別の場所にあるディレクトリを自分のホームディレクトリにコピーするにはどうすればよいでしょうか？`/home2/mori2021/TA2021/`というディレクトリに `exercise` というディレクトリが置いてあり、その中には今日使うファイルの一部が入っています。これを自分のホームディレクトリにコピーしてきてください。なお、ファイル・ディレクトリの作成・削除、移動の仕方などが分からない人は TA の人に聞きながら練習してみてください。

## 1.2 リダイレクトとパイプ

さて次も前回習った話です。パイプは UNIX の哲学の根幹にあるという話でした。使えるようになっているでしょうか？ **標準出力をファイルに書き出したり、ファイルの中身を標準入力として読み込んだりする際に使うのがリダイレクト、標準出力を別の操作の標準入力として送り込むのがパイプ**でした。それぞれについて、前回の講義で簡単な例を用いて学習したと思います。ここでは、`head` と `tail` というコマンドを使って、パイプの使い方を少し練習してみましょう。

先ほどコピーしてきてもらったディレクトリに `students.txt` というファイルが入っていたはずです。ファイルの中に

は皆さんのログインネームと名前が学生証番号の順に並んでいます。これを開いてみましょう。まずホームディレクトリから`cd exercise`でディレクトリを移動して、`cat -n students.txt`としてみてください (`-n` は行番号を付加するオプションです)。

```
1  s172601 Hiromasa Akadama
.
.
.
```

上のような出力が得られるはずです。ターミナルからはみ出して見づらい場合は、前回習ったようにパイプで `less` などに送りましょう。`cat -n students.txt | less`です。

さて、ファイルの中身を見ることができたら、今度は自分の名前の行だけ抜き出してターミナルに表示させてください。方法はいろいろとあるかと思いますが、ここでは `head` と `tail` というコマンドとパイプを用いてみましょう。`cat`、`head`、`tail` の基本的な使い方は Table 1.2 の通りです。オプションなどの詳しい情報は `man` で調べてください。

表 1.2　ファイルの中身を出力する基本的なコマンドとその使い方

| コマンド名 | 出力 | 基本的な書式[*1] |
|---|---|---|
| `cat` | ファイルの中身 | `cat <file>` |
| `head` | ファイルの先頭から指定した行数 | `head` *-lines* `<file>` |
| `tail` | ファイルの末尾から指定した行数 | `tail` *-lines* `<file>` |

さて、解答編です。例えば、4 行目を抜き出したいとします。`head -4 students.txt`だと、上から 4 行が出力されてしまいます。`tail -20 students.txt`だと、下から 20 行です。自分の名前が 1 行目や最後の行にある人は良いですが、これでは任意の行を抜き出すことができません。そこで、`head -4 students.txt`の出力をながめてみましょう。

```
s172601 Hiromasa Akadama
s172602 Kosuke Ikehata
s172603 Kensuke Ishikawa
s172604 Sohei Ishizuka
```

この出力の一番最後の行を抜き出してやれば、`students.txt` の 1 行目のみを出力することができますね。最後の行を抜き出すには`tail -1`としてやればよいので、あとはパイプで`head -4 students.txt`の出力を送ってやればよいわけですね。

```
~/exercise$ head -4 students.txt | tail -1
```

または少し長くなってしまうやり方ですが、`cat` も使って

```
~/exercise$ cat students.txt | head -4 | tail -1
```

としても同じ結果になるはずです。

機能としては、先頭から任意の行数を抜き出すコマンドと末尾から任意の行数を抜き出すコマンドがそれぞれ存在する

だけですが、パイプを使って組み合わせることで任意の行を抜き出すことができるツールになるわけです。任意の行を抜き出すコマンドを作らず、簡単な道具の組み合わせによって必要な処理を実現させる。それぞれの簡単な道具は、パイプという装置を通して、必要かつ十分な機能を持っており、それが **KISS (Keep It Simple and Smart)** を実現しているわけです。これが、パイプが UNIX の哲学の根幹と言われるゆえんです。

さて、任意の行が取り出せたので、次は任意の列を取り出す方法を見てみましょう。

`cat students.txt | cut -f 1,3 -d " "`と打ってみましょう（ダブルクオーテーション" " の間は半角スペースです)。すると

```
s172601 Akadama
s172602 Ikehata
.
.
.
```

のように、1 列目と 3 列目のみが表示されたはずです。ここで行った操作は、cut コマンドを使って半角スペースで区切られた（d オプション）1 列目と 3 列目を（f オプション）切り出したということです。cut コマンドは他にもいろいろな便利なオプションがついているので、調べてみるとよいでしょう。

### 1.2.1 練習

`students.txt` の 6 行目から 10 行を抜き出してみましょう。また、6 行目から 10 行目と 16 行目から 20 行目を `exercise.txt` というファイルに書き出してみてください。さらにそこから名前部分だけを `exercise2.txt` に書き出してみてください。リダイレクトの '>' と '>>' の違いに気をつけて、操作を行いましょう。

この程度の作業であれば、わざわざターミナル上で行わなくても Emacs などのエディタで十分だと思うかもしれませんが、本格的に計算を始めると、100 個、1000 個という規模のデータファイルを扱う機会も出てきます。Bash などのシェルを用いた、さらに高度な操作については ~~5 月後半の講義~~でシェルスクリプトというものを習います。特殊な文法などもありますが、基本はターミナル上でできることを組み合わせて作るプログラムのようなものになります。

コマンドというと何やら難しそうですが、イメージとしてはちょっとしたことができるプログラム群と考えれば良いわけです。`ls` はカレントディレクトリのファイルやディレクトリを表示してくれます。`head` や `tail` はファイルの先頭や末尾から指定した行だけ抜き出してくれます。パイプやリダイレクトを利用することで、処理を重ねて行わせて複雑なことをしたり、結果をファイルに書き出したりすることもできます。**「一つ一つはちょっとしたこと、しかし組み合わせることでいろいろなことができる」**、この辺りは UNIX の哲学に関係する部分です。詳しくは前回のレジュメを読み直してみてください。

# 第 2 章

# UNIX をもっと詳しく

## 2.1　シェルの基本機能

### 2.1.1　シェルとは

OS（Operating System）とは、コンピュータのハードウェア（CPU・メモリ・ストレージ・ネットワークカードなど）を制御し、人間（ユーザー）やアプリケーションソフトウェア（アプリ）に利用環境を提供するソフトウェアです。

そしてシェルとは、OS と人間が対話するためのインターフェイスです。ユーザーから「コマンド」を受け取り、機械語に翻訳して OS の中核「カーネル」に伝達します。そしてその実行結果を受け取って、人間がわかるように出力します。[*1]

シェルにはコマンドを 1 つずつ対話的に実行する「対話モード」と、スクリプトに書いて一括処理する「バッチモード」があります。「バッチモード」では Fortran のようにより複雑な処理を行うことができます。Linux では Bash というシェルがデフォルトで使われていることが多いです。この資料でも Bash を前提に解説します。

### 2.1.2　変数

シェルでも Fortran や Python のように変数を定義することができます。といっても Fortran のように型を気にする必要はありません（変数の型はコマンドによって都合よく解釈されます）。単純に

```
~$ TORI=inko
```

と入力すると、TORI という変数に inko という値が格納されます。（注意：=の前後にスペースを入れてはいけません！）変数名として使用できるのは英数字と _ のみで、先頭に数字は使えません。また Fortran と違い、大文字と小文字は区別されます（TORI と tori と ToRi は別の変数になります）。変数に格納された値は、$を変数名の前につけるか、${ }で変数名を囲むことで参照できます。すなわち

```
~$ echo $TORI
inko
```

または

```
~$ echo ${TORI}
inko
```

[*1] Windows であれば Explorer やスタートメニューなど、MacOS であれば Finder や Dock などがシェルといえます。（これらはグラフィカルシェルとも呼ばれます）

のようにします。また、これは表示されますが

```
~$ echo tobu$TORI
tobuinko
```

これは何も表示されません。

```
~$ echo $TORItobu
```

変数 TORItobu の値を表示しようとして、何も格納されていないので何も表示されなかった、ということです。このような場合は

```
~$ echo ${TORI}tobu
inkotobu
```

のように変数名を明示的に${ }で囲む必要があります。

### 2.1.3 ファイルディスクリプタとリダイレクション

UNIX では、ディスクやストレージ、キーボード、画面なども「ファイル」として扱います。ファイルディスクリプタとは、プログラムが操作するファイルを識別するために割り当てられる識別子です。通常、ファイルディスクリプタの 0 番は「標準入力（stdin)」としてキーボードに、1 番は「標準出力（stdout)」として画面に、2 番は「標準エラー出力（stderr)」として画面に割り当てられます。そしてプログラムがファイルに読み書きする際には、ファイルディスクリプタが 3 番から順に割り当てられていきます。

しかし、時にはキーボードではなくファイルから入力したり、画面ではなくファイルに出力したりしたい場合もあります。そのためにはファイルディスクリプタの指す対象を変更する必要があり、この操作を「リダイレクション」といいます。リダイレクションの方法を見ていきましょう。上から順に実行してください。

| 具体例 | 説明 |
|---|---|
| `echo $TORI > bird.txt` | 標準出力をファイルに上書きする。 |
| `echo tobu$TORI >> bird.txt` | 標準出力をファイルに追記する。 |
| `ls $TORI 2> error.txt` | 標準エラー出力をファイルに上書きする。 |
| `ls tobu$TORI 2>> error.txt` | 標準エラー出力をファイルに追記する。 |
| `echo $TORI 1>&2` | 標準出力を標準エラー出力に向ける。 |
| `ls $TORI 2>&1` | 標準エラー出力を標準出力に向ける。 |
| `wc -l < bird.txt` | ファイルの内容を標準入力にする。 |

表 2.1: リダイレクション

**練習問題 1**

上から順に実行して、出力ファイルの内容を確認してください。

もう少し具体的な例を考えます。標準出力と標準エラー出力をするプログラムがあるとします。[*2]

[*2] 計算機演習の課題でこのようなプログラムを書くことはないと思いますが、大規模なプログラムでは Fortran に限らずよくあることです。

Listing 2.1　stdouterr.py

```
1   import sys
2   import time
3   sys.stdout.write('1: This is stdout.')
4   sys.stdout.flush()
5   time.sleep(0.1)
6   sys.stdout.write('2: This is stdout.')
7   sys.stdout.flush()
8   time.sleep(0.1)
9   sys.stderr.write('3: This is stderr.')
10  sys.stderr.flush()
```

これを python で実行してみます。

```
~$ python stdouterr.py
1: This is stdout.
2: This is stdout.
3: This is stderr.
```

「標準出力はファイルに出力するが、エラーはファイルにも画面にも出力させない」という場合は、このようにします。

```
~$ python stdouterr.py > output.txt 2> /dev/null
~$ cat output.txt
1: This is stdout.
2: This is stdout.
```

「/dev/null」とはなんでしょうか。「/dev」というディレクトリには、ディスクやストレージなど様々なデバイスが「ファイル」として存在しています。そしてその中には「疑似デバイス」[*3]と呼ばれる仮想的なデバイスもあります。「/dev/null」はそのひとつで、あらゆる入力を捨て、何も出力しないデバイスです（ブラックホールのようなものです）。標準エラー出力は要らないので/dev/null に捨てた、ということです。

では、標準出力と標準エラー出力を同じファイルに順番通り出力するには、どうしたらよいでしょうか。例えばこのようにすればよいと思った人もいるでしょう。

```
~$ python stdouterr.py > output.txt 2> output.txt
~$ cat output.txt
3: This is stderr.
2: This is stdout.
```

[*3] 疑似デバイスには他に、ひたすら 0 を出力する「/dev/zero」、ひたすら乱数を出力する「/dev/random」があります。たとえば USB メモリのデータをすべて完全に消去するために、フォーマットしたあと

```
$ cat /dev/zero > /mnt/usb/zero.dat+
```

などという使い方をします。ただし、ファイル出力すると一瞬で DISK FULL になり、標準出力するとシステムが不安定になるので、少なくとも授業中は試さないでください。(振りではありません)

コマンドは一見よさそうです。しかし出力ファイルを見てみると、標準出力と標準エラー出力がぐちゃぐちゃになっていて、期待通りにファイルに書き込まれていません。[*4]正しく出力するためには、このようにします。

```
~$ python stdouterr.py > output.txt 2>&1
~$ cat output.txt
1: This is stdout.
2: This is stdout.
3: This is stderr.
```

シェルは左から順に解釈します。まず「`> output.txt`」で標準出力が output.txt にリダイレクトされ、次に「`2>&1`」で標準エラー出力が標準出力と同じファイルディスクリプタ、すなわち output.txt にリダイレクトされる、ということです。このようにして標準出力と標準エラー出力を順番通り同じファイルに出力することができました。

ではこのようにするとどうなるでしょうか。

```
~$ python stdouterr.py 2>&1 > output.txt
3: This is stderr.
~$ cat output.txt
1: This is stdout.
2: This is stdout.
```

これは、まず「`2>&1`」で標準エラー出力が標準出力と同じファイルディスクリプタ、すなわち画面にリダイレクトされ、次に「`> output.txt`」で標準出力が output.txt にリダイレクトされます。従ってファイルに書き込まれるのは標準出力だけになるのです。この辺はシェルの紛らわしいところです。

なお bash では、単に

```
python stdouterr.py &> output.txt
```

としても標準出力と標準エラー出力を同じファイルに出力することができます。

### 2.1.4 パイプ

パイプとは、標準出力をそのまま次のコマンドの標準入力に渡す仕組みです。asano で次のコマンドを実行しましょう。

```
~$ sleep 600 &
```

`&` をつけることで、`sleep` コマンド（何もしないで待機するコマンド）をバックグラウンドジョブとして実行しました。[*5]次に

```
~$ ps aux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root          1  0.0  0.0 178780  6680 ?        Ss   Apr01   0:48 /sbin/init
root          2  0.0  0.0      0     0 ?        S    Apr01   0:01 [kthreadd]
```

[*4] これについては、Fortran と Ruby で結果が違うなどプログラミング言語によっても挙動が異なっていて、また文献にもインターネットにもちゃんとした解説が見つからなかったため、推測になりますが、標準出力先と標準エラー出力先が独立したファイルと認識され、出力時にファイルのアクセス位置が正しくならない（重複してしまう）ためと考えられます。標準エラー出力を追記（`2>>`）にしてもうまくいきません。

[*5] emacs (GUI) でファイル編集したり、長い時間がかかる計算や処理をするときは、バックグラウンドジョブとして実行しましょう。

```
root           3  0.0  0.0      0     0 ?        S    Apr01  30:46 [ksoftirqd/0]
root           5  0.0  0.0      0     0 ?        S<   Apr01   0:00 [kworker/0:0H]
root           6  0.0  0.0      0     0 ?        S    Apr01   0:00 [kworker/u64:0]
...
```

とすると、asano で実行中のプロセスがすべて表示されます。しかしあまりに多すぎます。そこで、

```
~$ ps aux | grep 'sleep'
s212613   8408  0.0  0.0   5808  1408 pts/71   S    13:43   0:00 sleep 600
s212600   8418  0.0  0.0   5808  1360 pts/72   S    13:44   0:00 sleep 600
s212630   8423  0.0  0.0   5808  1348 pts/73   S    13:44   0:00 sleep 600
s212615   8434  0.0  0.0   5808  1352 pts/70   S    13:44   0:00 sleep 600
...
s212600   8437  0.0  0.0  12700  1780 pts/70   S+   13:45   0:00 grep sleep
```

とします。すると実行中の sleep のプロセスが表示されました。ただし「grep sleep」そのものも表示されてしまうので、これを消すにはこのようにします。

```
~$ ps aux | grep 'sleep' | grep -v 'grep'
s212613   8408  0.0  0.0   5808  1408 pts/71   S    13:43   0:00 sleep 600
s212600   8418  0.0  0.0   5808  1360 pts/72   S    13:44   0:00 sleep 600
s212630   8423  0.0  0.0   5808  1348 pts/73   S    13:44   0:00 sleep 600
s212615   8434  0.0  0.0   5808  1352 pts/70   S    13:44   0:00 sleep 600
...
```

ではいま sleep を実行しているのは何人でしょう。行数を数えると

```
~$ ps aux | grep “sleep” | grep -v “grep” | wc -l
32
```

これで asano で実行中の sleep のプロセスの数がわかりましたね。このように、パイプはどんどんつなげていくことができます。このように、単純な仕事をするコマンドをひとつひとつつなげて複雑な処理をする、というのが UNIX の根底にある考え方です。

ちなみに、標準エラー出力をパイプで渡すには、次のようにします。

```
~$ ls dir_parrot 2>&1 | rev
```

**練習問題**

自分が実行した sleep のプロセスを強制終了（kill）してください。

**練習問題**

wc -l bird.txt と wc -l < bird.txt と cat bird.txt | wc -l の違いを考えてください。（結果は同じです）

### 2.1.5　コマンドセパレータ

パイプと同じように、コマンドをつなげる仕組みとしてコマンドセパレータというものがあります。ただし、パイプと異なり、前のコマンドの出力を引き継ぐことはできません。

| コマンドセパレータ | 説明 |
|---|---|
| `command1 ; command2` | command1 が終了したら command2 を実行する。 |
| `command1 && command2` | command1 が成功（正常終了）したら command2 を実行する。 |
| `command1 \|\| command2` | command1 が失敗（異常終了）したら command2 を実行する。 |

表 2.2: コマンドセパレータ

例えば次のように使います。（結果は書きません）

```
~$ pwd ; ls
~$ ls bird.txt && cp bird.txt vorgel.txt
~$ ls dir_parrot || mkdir dir_parrot
```

### 2.1.6　メタキャラクタと特殊文字

シェルのメタキャラクタと特殊文字をまとめます。

| 具体例 | 説明 |
|---|---|
| ? | 任意の 1 文字。 |
| * | 任意の文字列。 |
| [　] | 角括弧内のいずれかの 1 文字である。 |
| [!　] | 角括弧内のどの 1 文字も含まない。 |
| - | 角括弧内で、連続する文字を指す。（例：a-z,0-9） |
| {　} | 波括弧内のいずれかの文字列である（, で区切る）。 |
| $var | 変数 var の値に置き換える。 |
| ~ | ホームディレクトリに置き換える。 |
| ‘command‘ | コマンド command の実行結果に置き換える。 |
| $(command) | コマンド command の実行結果に置き換える。 |
| command & | コマンド command をバックグラウンドジョブとして実行する。 |
| “　“ | クオーテーション内の$, \, ‘以外のメタキャラクタと特殊文字を無効化し、単なる文字として処理する。 |
| ‘　‘ | クオーテーション内のすべてのメタキャラクタと特殊文字を無効化し、単なる文字として処理する。 |
| \ | 直後のメタキャラクタと特殊文字を無効化する。 |
| # | コメントアウト |

表 2.3: シェルのメタキャラクタと特殊文字

## 2.2　コマンド呼び出しの仕組み

### 2.2.1　コマンドはどこからくるのか？

さて、コマンドがちょっとしたことができるプログラムということは良いでしょうか？ そう考えると、ここで少し疑問が出てきます。Windows でも Mac OS X でも、アプリケーションなどのプログラムはアプリケーションが入っている場所までいって起動をしたり、ショートカットなどから起動したりしますね。しかし`ls`をしても分かるように、皆さんのホームディレクトリにコマンドに関係しそうなファイルはありません。一体コマンドというもの達はどこからどのように呼び出されているのでしょうか？ ここから少し UNIX のシステムの話になります。

まず、前回の講義に載っていた UNIX における代表的なディレクトリとその中身という図をもう一度見てみましょう。

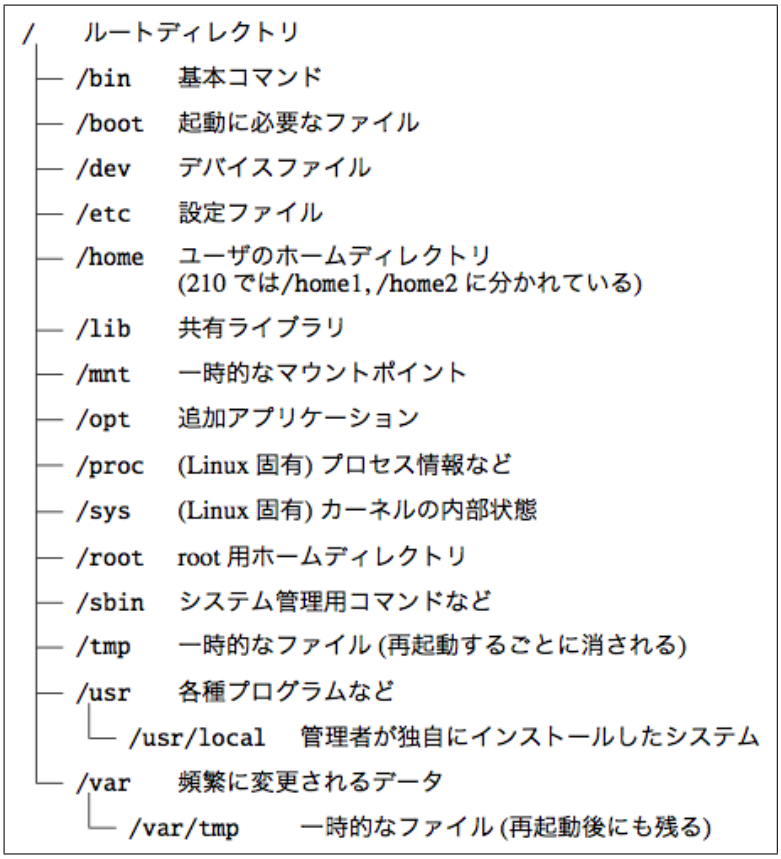



図 2.1　UNIX における代表的なディレクトリとその中身

Fig. 2.1 を見ると、基本コマンドは`/bin` にあるようです。`find /bin -name ls`としてみましょう。

```
~$ find /bin -name ls

/bin/ls
```

とお返事がいただけますね。どうやら`/bin` というディレクトリに `ls` が存在しているようです。`head` についても`/bin` にあるのでしょうか。同じように`find /bin -name head`としてみましょう。お返事が返ってこないのではないかと思いま

す。こういうときはwhereis headと入力してみましょう。

```
~$ whereis head

head: /usr/bin/head /usr/bin/X11/head /usr/share/man/man1/head.1.gz
```

というお返事が返ってくるかと思います。後ろにいろいろと付いているかと思いますが、いまは気にせずに流してください。どうやらheadは/usr/binにあるようですね。Fig 1 を見てみると、/usrは各種プログラムなどが入っているディレクトリです。ちなみに、man whereisとすれば分かりますが、whereisはコマンドのバイナリ・ソース・man ページの場所を示してくれるコマンドです。その他のコマンドについても、いくつか試してみましょう。

### 2.2.2 コマンドはどうやってくるのか？

さて、コマンドがどこにいるのかは分かってきました。どうやらlsは/binというディレクトリにいて、headは/usr/binにいるようです。試しに、/bin/ls -lと入力してみましょう。

```
~$ /bin/ls -l
```

すると、lsと全く同じ出力が得られるはずです。場所さえ知っていれば、このように使うことができるのは Windows や Mac OS X などのアプリケーションと同じですね。

しかし、先ほどいくつかのコマンドで確認したように、皆が同じ場所にいるわけではないようです。にも関わらず、我々はコマンドがどこにいるのかを全く知らないまま使うことができています。不思議ですね。この節では、コマンドが呼び出される仕組みについて説明します。

ヒントは、ログイン時に読み込まれるファイル/etc/profileにあります。起動時のファイル読み込みについては後ほど詳しく説明しますが、とりあえず今は中を確かめてみましょう。ターミナル上でless /etc/profileと入力してみてください。何やら暗号めいた内容が表示されるかと思いますが、一つ一つの解読はそれぞれの興味にお任せします。各自調べてみたり、TA に聞いてみたりしてみてください。ここでは、先ほどからコマンドのありかとして現れているbinという単語を検索してみましょう。~~less~~ や less 内での検索には/<search term> ↵ と入力します (つまりスラッシュを打った後探したい言葉を入力する、n と打つと次の検索結果に移る)。この場合、/bin ↵ です。最初の方の PATH=... と書かれた箇所を探してください。

```
if [ "`id -u`" -eq 0 ]; then
  PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
else
  PATH="/usr/local/bin:/usr/bin:/bin:/usr/games"
fi
```

このような記述が見つかりましたか？ さらにこのような記述が最後の方にあったと思います。

```
export PATH
```

export は bash において環境変数を設定するコマンドです。そしてこの**環境変数** PATH が、使えるコマンドの場所を指定しています。つまり、いちいちそのコマンドの置き場を指定しなくても、PATH で指定されているディレクトリについては自動的に探してくれるということです。“:” で区切られて、いくつかのディレクトリが設定されていますが、前に書かれているものほど**優先的**に呼び出されます。つまり、上の例では、仮に/usr/local/bin/ls と/bin/ls が存在していた場合、普通に ls と入力したときには前の方にある/usr/local/bin/ls が実行されるということです。環境変数 PATH において設定され、そのディレクトリのコマンドなどをパスの指定なしに利用できる状態のことを**パスが通っている**というような言い方をします。CUI に限らず、計算機を便利に利用するためには、パスを通すという概念が非常に重要です。現在パスが通っているディレクトリがどうなっているかは、環境変数 PATH を表示する以下のコマンド

```
~$ echo $PATH
```

を実行すれば確認できます。すると、(ちょっと見づらいかもしれませんが) 確かに/usr/local/bin:が入っていることがわかると思います。先ほどの ls と head について、上の PATH で確認してみましょう。今は下段の方の PATH の設定が効いています。確かに/bin にも/usr/bin にもパスが通っていますので、ls も head も取り立ててパスの指定をすることなく利用できていたわけですね。

ではここで少し試してみましょう。先ほどの~/exercise ディレクトリに line.sh というコマンドを用意しておきました。とりあえず拡張子.sh については気にしないでください。これは cat コマンドなどで中をみてみれば何となくわかると思いますが、要するに<ファイル名>の<整数> 行目を抜き出してください、という意味です。適当に試してみましょう。

```
~$ ~/exercise/line.sh 10 ~/exercise/students.txt
```

お返事がありましたよね？ でも

```
~$ line.sh 10 ~/exercise/students.txt
```

とだけしても、そんなコマンドないよって怒られるはずです。たとえ今いるディレクトリに line.sh が入っていてもやはり実行できません。そこで

```
~$ PATH=$PATH:~/exercise

~$ line.sh 10 ~/exercise/students.txt
```

としてみましょう。今度は怒られないはず。~/exercise にパスを通すことにより（これまでのパスに~/exercise を追加）line.sh コマンドが使えるようになったわけです。

### 2.2.3 環境変数について

環境変数について、少しだけ説明しておきます。環境変数とは、その名の通り、端末の環境に関する情報です。ターミナルでenv (environment の略) と入力してみて下さい。

```
MANPATH=/usr/local/intel/idb/man:/usr/local/intel/ifc/man:/usr/local/intel/icc/man:/usr/local/
intel/idb/man:/usr/local/intel/ifc/man:/usr/local/intel/icc/man:/usr/local/intel/idb/man:/usr/
local/intel/ifc/man:/usr/local/intel/icc/man:/usr/local/intel/icc/man:/usr/local/man:/usr/loca
l/share/man:/usr/share/man
```

```
INTEL_LICENSE_FILE=/usr/local/intel/icc/licenses:/opt/intel/licenses:/home2/koshin/intel/licen
ses:/Users/Shared/Library/Application Support/Intel/Licenses:/usr/local/intel/ifc/licenses:/op
t/intel/licenses:/home2/koshin/intel/licenses:/Users/Shared/Library/Application Support/Intel/
Licenses:/usr/local/intel/icc/licenses:/opt/intel/licenses:/home2/koshin/intel/licenses:/Users
/Shared/Library/Application Support/Intel/Licenses:/usr/local/intel/ifc/licenses:/opt/intel/li
censes:/home2/koshin/intel/licenses:/Users/Shared/Library/Application Support/Intel/Licenses:/
usr/local/intel/icc/licenses:/opt/intel/licenses:/home2/koshin/intel/licenses:/Users/Shared/Li
brary/Application Support/Intel/Licenses:/usr/local/intel/ifc/licenses:/opt/intel/licenses:/ho
me2/koshin/intel/licenses:/Users/Shared/Library/Application Support/Intel/Licenses
TERM=xterm-256color
SHELL=/bin/bash
.
.
.
```

このようにさまざまな環境変数があります。これらの環境変数が、現在皆さんが使っている端末において定義され、端末の環境を規定しているわけです。今後色々なアプリケーションをインストールするとき、インストールガイドなどで「○○という環境変数を設定するように」という指定がなされることがあると思います。そのときはここでやったことを思い出してみるとよいでしょう。また、興味のある方は、それぞれの意味について調べてみると良いと思います。

## 2.3 起動のはなし

さて、話を本筋に戻しましょう。これまでの話をまとめると、**コマンドは主に /bin や /usr/bin などのディレクトリ内にしまわれており、それらのディレクトリにパスを通すことによって、パスを指定しなくてもコマンドが利用できる**という話でした。**環境変数 PATH は /etc/profile の中で定義されていました**。ここまでは良いですね。

少しずつ、端末の中がどうなっているのかが分かってきましたね。しかし、まだ大きな疑問が残っています。本来パスの設定などは個人設定として、自分で自由に追加したり削除したりしたいところです。でも先ほどの/etc/profile というディレクトリは、いかにも個人で変えられそうもないところにあります。実際にemacs /etc/profile &などとして何か書きかえようとしてみればわかります。権限がないと怒られるはずです。ではどのようにすれば自分で好きなように個人用の環境変数を設定すればよいのでしょうか。それを理解してもらうために、この章では、端末の起動の仕組みについて説明します[*6]。

### 2.3.1 ログイン

もう問題無いかと思いますが、ユーザ名とパスワードを入力して、端末にログインをします。ここでユーザの認証が行われて、それぞれの環境に向かい入れてもらえます。ちなみに現在の 210 はパスワード入力時に * が表示されると思いますが、UNIX 系の端末ではパスワードの流出を防ぐため、何も表示されないこともあります。ネットワークの回で学んだ ssh を用いて、外部ログインをすでに試した人は、ログインするときパスワード入力のとき何も表示されないことに気づいたと思います。

[*6] 使っているシェルやディストリビューションによって、異なる部分がありますが、ここでは 559 の環境を想定し、典型的な一例を示します。

### 2.3.2 /etc/profile の読み込み

ログインに成功すると、bash が起動し、/etc/profile が読み込まれます。これは全てのユーザに共通の設定ファイルであり、ログイン時に一度だけ読み込まれます。

### 2.3.3 ~/.profile の読み込み

/etc/profile の次に読み込まれるのが、ホームディレクトリにある.profile[*7]です。ホームディレクトリに存在することからも分かるように、このファイルは個人用の設定ファイルです。.profile も**ログイン時に一度しか**読み込まれません。/etc/profile との違いは、**全ユーザに共通か、個人にのみ適用されるか**、という点です。つまり、全ユーザについて設定しておくべきことは/etc/profile に書き込まれます。

### 2.3.4 ~/.bashrc の読み込み

次に読み込まれるのが、.bashrc です。.bashrc は**シェルを起動する毎 (ターミナルなどを新たに立ち上げる毎)** に読み込まれます。もちろん、これも個人毎の設定ファイルになります。.profile とのもっとも大きな違いは、**.bash_profile がログイン時に一度だけ読み込まれるのに対し、.bashrc はシェルを起動するたびに読み込まれる**という点です。つまり、一度だけ設定すればよいことは.profile に書かれ、シェルを起動する毎に設定すべきことが.bashrc に書かれます。先ほどの PATH の設定などは.profile と.bashrc のどちらに書くこともできます. ただ、.bashrc の中で設定した方がターミナルを立ち上げ直すだけですぐに変更が反映される (いちいち再ログインしなくてよい) ので、そちらの方が一般的です。皆さんの場合は

```
# PATH etc の設定
export PATH=/usr/local/intel/idb/bin:/usr/local/intel/ifc/bin:
/usr/local/intel/icc/bin:/usr/local/bin:/usr/bin:/bin:/usr/games
```

などとしておいて[*8]これに適宜足していくのがよいかと思います。また、一度に書かずに

```
# PATH etc の追加
export PATH=$PATH:<追加したいパス>
```

などとして、これまでのパスに順次追加することもできます。

その他にも、/etc/.profile.d というディレクトリや、~/.bash_login、~/.bash_logout といったファイルが存在する場合もありますが、概ねの流れは 3.1 節から 3.4 節のような感じです。Fig 2.2 に流れを示しておきます。

また、予想できるかと思いますが、.bash_history はコマンドのヒストリ機能を助けるためのファイルです。less .bash_historyで中をのぞいてみると、皆さんがこれまでに入力したコマンドが並んでいるはずです。

[*7] .bash_profile のような名前になっている場合もある。

[*8] 見やすさのために改行していますが、実際にはこの書き方はできません。1 行で書くか、改行するところにバックスラッシュを付ける必要があります。

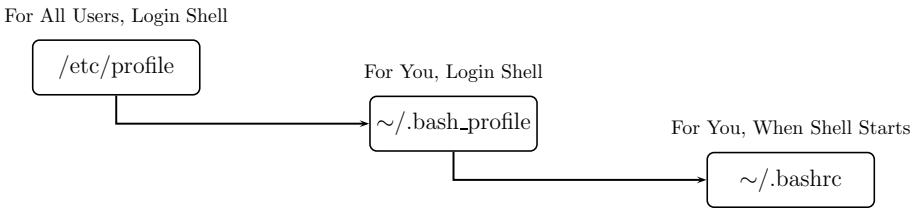



図 2.2　ログイン後に読み込まれるファイル

### 2.3.5　ファイルの中をのぞいてみよう

さて、今までに出てきたファイルの中身を少しのぞいてみましょう。

まず、/etc/profile をのぞいてみましょう。less /etc/profileとしてください。

```
 1 # /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
 2 # and Bourne compatible shells (bash(1), ksh(1), ash(1), ...).
 3
 4 if [ "`id -u`" -eq 0 ]; then
 5   PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"
 6 else
 7   PATH="/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games"
 8 fi
 9 export PATH
10
11 if [ "$PS1" ]; then
12   if [ "$BASH" ] && [ "$BASH" != "/bin/sh" ]; then
13     # The file bash.bashrc already sets the default PS1.
14     # PS1='\h:\w\$ '
15     if [ -f /etc/bash.bashrc ]; then
16       . /etc/bash.bashrc
17     fi
18   else
19     if [ "`id -u`" -eq 0 ]; then
20       PS1='# '
21     else
22       PS1='$ '
23     fi
24   fi
25 fi
26
```

```
27 # The default umask is now handled by pam_umask.
28 # See pam_umask(8) and /etc/login.defs.
29
30 if [ -d /etc/profile.d ]; then
31   for i in /etc/profile.d/*.sh; do
32     if [ -r $i ]; then
33       . $i
34     fi
35   done
36   unset i
37 fi
```

シェルスクリプトについては、後では触れますが、意味の分かる部分も出てきたかと思います。4 行目の if はプログラミングをやったことがある人なら分かるかもしれませんが、条件によって異なる処理を行う際に用いられる構文です。その後の id という部分は、man idとしてみましょう。どうやらユーザの情報を参照して、それによって処理を変えているようです。ユーザの種類によって、PATH の値を変更しているようです。PS1 に関しては、後でもう少し詳しく説明します。というように、何となく意味が分かっていくかと思います。繰り返しになりますが、9 行目の export というのは、シェルスクリプトの中で定義したシェル変数を環境変数にするという操作をしています。少しややこしいですが、export をしないと、代入された値などはそのシェルの中でしか定義されたことにならず、export が行われることで環境変数として、そのプロセス (ここでは/etc/profile が実行されたシェル内) から起動されるプロセス (例えば、新たに立ち上げたシェル) にも値が引き継がれるということになります。

次は~/.bash_profile の中をのぞいてみましょう。先ほどと同様、less  /.bash_profileとしてください (文字化けするときはless /etc/profile | nkf -w としてみてください)。

```
1  # ---- language-env DON'T MODIFY THIS LINE!
2  # .bash_profile は、ログイン時に実行される。
3  if [ -f ~/.bashrc ]
4  then
5    # ただし、すでに .bash_profile が .bashrc を実行していたら、
6    # 重複しては実行しない。
7    if [ -z "$BASHRC_DONE" ]
8    then
9      . ~/.bashrc
10   fi
11 fi
12 # ---- language-env end DON'T MODIFY THIS LINE!
```

コメントの通り、.bash_profile はログイン時に実行されます。3 行目は.bashrc が存在するかを確かめる条件文です。コメントを付けているので大体分かるかと思いますが、コメント (# 印) の 5 行目と 6 行目に書かれている内容が

7 ～ 11 行目に書かれているという感じです。しかし実はこの部分は役に立っていません。それについては後ほど。

内容としては、.bashrc を実行してね、ということですね。皆さんの.profile は条件付き (.bashrc の有無や、すでに実行されたかどうか) で実行するように調整されていますが、人によっては、.profile の記述は、

```
source ~/.bashrc
```

のみという人もいます。

ここまでがログイン時に設定されることです。つまり、もし.bashrc という個人設定ファイルがなくても、/etc/profile の中に書かれていたことは設定された状態になるということです。PATH や umask などの基本的な部分はここまでですでに定義されていますね。.profile を見れば分かるように、別に.bashrc が無くても、端末が起動しないわけではありません。なので、自分が新たにパソコンを使い始める時に.bashrc がないからといって焦る必要はありません。.bashrc は基本的には自分で作るものなので、もしなければ新たに作ればよいというだけです. .bashrc の中での設定は、端末を便利に使う上で非常に重要なものばかりです。それでは中をのぞいてみましょう。.bashrc はそれなりに内容が多いので、今回はその中の一部を説明します。行数を利用して説明しますので、cat -n .bashrc | lvなどとしてください[*9]。その後、何となく分かるような分からないような記述が続きますが、最後までみても先ほどの BASHRC_DONE という環境変数が現れないことが分かると思います。.bashrc を起動したか否かを判断するための変数なのに.bashrc の中に記述がない。これでは.profile でわざわざ重複を避けている意味がありませんね。役に立っていないと言ったのはそういう理由です。気になる人は

```
# .bash_profile で使う。
 BASHRC_DONE=1
```

などと足しておけばいいでしょう (別にこれをやらなくても何も問題はありません)。

11 ～ 20 行目の PS1 については、後の練習で利用します。下に進んでいくと、27 行目からは、aliases という名前がみえると思います。

```
28  if [ -f ~/.aliases ]; then
29      .  ~/.aliases
30  fi
```

エイリアスというのは覚えていますか？.aliases に関しては後で説明しますが、どうやらホームディレクトリ下の.aliases の存在を確認し、読み込んでいるようです。

また 22 行目に書かれている/etc/bash_completion はシェルの補完機能[*10]を拡張するためのファイルです。補完機能の拡張は端末を便利に使うためには不可欠です。興味があれば、中をのぞいてみてください。

その他、気になる部分は各自調べてみたりしましょう。おそらく、最初に自分で設定ファイルを作る際は、ネット上に転がっているファイルを拾ってきて、カスタマイズしていったり、便利そうな機能をコピペしていくという形になるのではないかと思います。その時のためにも暗号解読のスキルを身に付けておきましょう。

---

[*9] もう意味は分かりますよね！

[*10] Tab を押すとコマンド名やらファイル名やらが補完される機能。ちゃんと使っていますよね。使わないと非常に不便です。

### 2.3.6 練習

.bashrc に現れる環境変数を少しいじってみましょう。11 行目からの PS1 という変数を使って、遊んでみます。まずは、.bashrc の中身を確認しておきます;

```
11  # プロンプト。man bash 参照
12  if [ "$TERM" = "dumb" -o "$TERM" = "emacs" ]; then
13    PS1='\w\$ '
14  else
15    if [ "$UID" = "0" ]; then
16      PS1='\[\e[41m\]\w\$\[\e[m\] '
17    else
18      PS1='\[\e[7m\]\w\$\[\e[m\] '
19    fi
20  fi
```

PS1 はプロンプト (コマンドを打ち込む前に表示される印) を定義する変数です。どこのことを言っているのか分からない人もいると思いますので、とりあえずターミナル上で次のように入力してみましょう。'=' の前後に空白などは入れないでください。> の後ろには空白を 1 つくらい入れておいた方が、おそらく幸せです。

```
~$ PS1='> '
```

PS1 とは、これによって形が変わった場所のことです。良いですか？ これでは変で使いづらいとなったら

```
> source .bashrc
```

とすれば元に戻せます。注意して欲しいのは、「PS1」という変数は export されていないので、環境変数ではないということです。このことは

```
~$ env
```

としてみても、PS1 は現れないということから確認できます。

ここからは少し遊んでみましょう。以下の例を実行してみて下さい。また、Table 2.4 にいろいろな情報の参照の仕方をまとめました。いろいろと遊んでみましょう。

```
~$ PS1='\h '
```

```
~$ PS1='\h \u \n \d \t '
```

元祖ニコちゃんマーク。

```
~$ PS1=':-) '
```

冷や汗。

```
~$ PS1='^^; '
```

他にも色々バリエーションはあると思います。色なども設定できます。

表 2.4　PS1 で利用できる情報とその表記の例

| 表記 | 意味 |
|---|---|
| \h | ホスト名 |
| \u | ユーザ名 |
| \d | 日付 |
| \t | 時刻 (24 時間制) |
| \T | 時刻 (12 時間制) |
| \w | カレントディレクトリ (フルパス) |
| \W | カレントディレクトリ名 |
| \n | 改行 |

## 2.4　設定ファイルやディレクトリ

今までに見てきた設定ファイルは、主にシェルに関わる設定ファイルでした。そのほかにもさまざまな設定ファイルが存在し、環境やアプリケーションのカスタマイズが行えるようになっています。

### 2.4.1　.ssh ディレクトリ

~/.ssh/config というファイルが SSH の設定ファイルです。たとえば、

```
Host dover
        HostName dover.eps.s.u-tokyo.ac.jp
        User s212601

Host edu01
        HostName edu01.eps.s.u-tokyo.ac.jp
        User s212601
        ProxyCommand ssh -X dover nc %h %p

Host www-geoph
        HostName www-geoph.eps.s.u-tokyo.ac.jp
        User s212601
        ProxyCommand ssh -X dover nc %h %p
```

のように記述しておくと便利です。こうしておけば

```
$ ssh -X s212601@dover.eps.s.u-tokyo.ac.jp
```

とわざわざ入力しなくても

```
$ ssh -X dover
```

と入力するだけで済むようになります。また edu や www-geoph にログインする場合も dover に一旦 SSH する必要はな

く、直接

```
$ ssh -X edu01
```

とするだけで済みます。SCP や SSHFS[*11] にも有効なのでぜひ設定しましょう。

### 踏み台サーバー

計算機演習室のサーバーは、dover 以外は学外から ssh できない設定になっています。そのため、みなさんは毎回最初に dover にログインしてから更に ssh して目当てのサーバーにアクセスする必要がありました。世の中にはこのような構成になっているサーバーがたくさんあります。こういったサーバーにアクセスする際に毎回 2 回の ssh ログインをし 2 回の exit をするのは手間です。これは実は設定ファイルに ProxyCommand という 1 行を書くだけで一発で認証できるようになります。

### 公開鍵認証

~/.ssh/には config 以外のファイルも配置されます。その中で最も重要なものが、鍵ファイルです。

これまで皆さんは、SSH で遠隔ログインする際にパスワードの入力を求められてきました。これは「背後に人が立っていてパスワードを盗み見られる」といった脆弱性があります。また、単純にパスワードを打つのが面倒だという問題もあります。そこで、これを解決する方法の一つとして公開鍵認証[*12]が広く用いられています。

設定方法は次の通りです。まず、ローカルホストで~/.ssh/へ移動し、

```
/.ssh$ ssh-keygen
```

を実行します。すると

```
Generating public/private rsa key pair.
Enter file in which to save the key (/home/xxx/.ssh/id_rsa):
```

と表示されるので、鍵の名前を設定します。未入力で進むと id_rsa という名前が自動的に設定されます。このテキストでは鍵の名前は id_rsa とします。

次に、

```
Enter passphrase (empty for no passphrase):
```

と表示されます。多くの場合は空にしますが、盗難等を心配して鍵を開くためのさらなるパスワードを設定することもできます。入力すると確認されるので、同じ文字列を再度入力します。

すると、~/.ssh/に id_rsa というファイルと id_rsa.pub という 2 つのファイルが作成されます。見ての通り、id_rsa.pub が公開鍵ですので、これをサーバーに転送して使います。

```
ssh-copy-id -i ~/.ssh/id_rsa.pub username@hostname
```

---

[*11] 本テキストでは説明しません。リモートマシン上のファイルをローカルのファイルと同じように扱えるため便利なので中〜上級者は使ってみてください。

[*12] 暗号化に用いる鍵と復号化に用いる鍵が異なっているもの。この場合鍵ペアのうち暗号化に用いるものは公開しても問題ない。

とすれば公開鍵がサーバーに転送され適切に配置されます。

最後に、鍵認証で接続する設定を行います。ssh コマンドのオプションを用いて

```
ssh -i id_rsa username@hostname
```

として接続することも可能ですし、~/.ssh/config に

```
Host asano
        HostName asano.eps.s.u-tokyo.ac.jp
        User s212601
        ProxyCommand ssh -X dover nc %h %p
        IdentityFile ~/.ssh/id_rsa
        IdentitiesOnly yes
```

のように IdentityFile 行を追加することによって鍵を指定することも可能です。

空ファイルの作成

$ touch filename

### 2.4.2 .bashrc

先ほどから何度も出てきているファイルです。中身は先ほども見ましたので、ここでは省略します。

### 2.4.3 .aliases

先日の講義でエイリアスという機能を習いました。良く使うコマンドやオプションなどを便利に使うためのショートカットのようなものでしたね、.aliases は、旧 edu では先ほど見た.bashrc の最後の方で読み込まれていました。[*13]中をのぞいてみましょう。しつこいようですが、lv .aliasesです。

```
alias md='mkdir'
alias rd='rmdir'
if [ "$TERM" = "dumb" -o "$TERM" = "emacs" ]; then
  alias ls='ls -F'
else
  alias ls='ls -F --color=auto'
fi
alias lf='ls -F'
alias la='ls -a'
alias ll='ls -l'
alias l.='ls -ld .*'
alias rm='rm -i'
```

[*13] bashrc 内に直接 alias が書き込まれている場合もあるようです。見当たらない場合は新規作成するか bashrc に直接書き込みましょう。

```
alias   ..='cd ..'

alias   ggre='firefox http://www.google.co.jp'

alias grep='grep --color'
alias a2ps='a2psj'
alias wl='emacs -nw -f wl'
alias kterm='kterm -bg gray30 -fg white -cr yellow'

if [ -x /usr/bin/xdvi-ja ]; then
  alias xdvi='xdvi-ja'
fi

function xtitle() {
  /bin/echo -e "\033]0;$*\007\c"
}
```

今回はややこしい部分には触れませんので、エイリアスで定義されているコマンドをいろいろと使ってみましょう。`la` や `ll` などはすでに使っている人もいるかもしれません。よく使うオプションはエイリアスで定義しておくと便利です。また、`mkdir` のエイリアスとなっている `md` のように、少し長めのコマンドなどにもエイリアスを使うと便利です。一つ上のディレクトリに移動する`..` なども使い慣れると必須になります。先ほどの `line` コマンドもエイリアスの中で定義してやれば問題なく使えます。

エイリアスの意味や効用は分かりましたか？

### 2.4.4 .emacs

そのまんまの名前ですが、Emacs の設定ファイルです。`.bashrc` がシェルの起動時に読み込まれるように、`.emacs` は Emacs の起動時に読み込まれるファイルです中をのぞいてみましょう。こちらも長いので、工夫して見てください。

ファイル内では言語環境などの設定がされていると思います。210 の環境では気づきませんが、23 行目、27 行目辺りで特に必要の無いメッセージは表示されないように設定されていたりもします。57 行目からは X で立ち上げたとき[*14]の設定が書かれており、メニューバーなどの表示・非表示もここで扱われています。88 行目からは色の設定がされています。120 行目からの記述は、TEX や Fortran を習うときに気にしておくとよいでしょう。

ちなみに、気づいたかと思いますが、先ほどまでのシェル関連のファイルと違い、`.emacs` ではコメントアウトの記号は ';' です。他のファイルと同列のように並べましたが、今まで見てきたファイルはシェルによって実行されるファイルだったのに対し、`.emacs` は Emacs Lisp という Emacs 用のプログラミング言語で書かれています。Emacs Lisp は Emacs Lisp で、語り出すと大変なことになりますので、興味のある人は調べてみて下さい。

[*14] Emacs はターミナル上で立ち上げることも可能です。`emacs -nw`としてみましょう。

# 第 3 章

# 知っておくと便利なコマンド

## 3.1　ファイル転送

遠隔ログインを使うようになると、ただログインするだけでなく、ファイルの転送もしたくなります。

### 3.1.1　SCP

ファイルの転送を暗号化するには、SCP (Secure CoPy) [*1]を用います。SSH を利用して通信を行う方式なのでセキュアです。dover や演習室の端末と自分のパソコン間でファイルをやり取りする場合にはこれを使うことになるので、使用頻度はかなり高いと思います。使い方は以下の通りです。

ローカルからリモートにファイルを送る場合には

```
$ scp filename1 [username@]hostname:filename2
```

`[username@]` となっているのは、ユーザ名がローカルとリモートで同じ場合に `username@` を省略できるということで、SSH の時と同様です。例えば、手元にあるレポート課題を dover に送りたい時は

```
$ scp report.pdf dover.eps.s.u-tokyo.ac.jp:enshu_report.pdf
```

のようになります。

反対にリモートからローカルにファイルをとってくる場合には

```
$ scp [username@]hostname:filename1  filename2
```

とします。さらに、-r オプションをつけると、ディレクトリのコピーもできます。recursive（再帰的）の r です。

いずれにせよ、「送るもの」を先に指定してから、「送り先」を指定します。前回扱った cp コマンドとよく似ていますね。

[*1] 2019 年 4 月より OpenSSH 公式より SCP は非推奨と宣言されています。本講義では cp コマンドとの類似性から引き続き SCP を用いてファイル転送の説明をしますが、余裕のある方は代替である sftp や rsync を調べてみてください。

### 3.1.2 練習

1. hoge1.dat というファイルを dover に送り、hoge2.dat という名前で保存してください。
2. 1. で送ったファイルを edu に取ってきて、hoge3.dat という名前で保存してください。

### 3.1.3 WGET

例えば、`http://www.eps.s.u-tokyo.ac.jp/access.html` というファイルをダウンロードしたいとき、どのような方法があるでしょうか。GUI が好きであれば Firefox で右クリックして保存しても良いですが…CUI も使えるようにして欲しいですね…。

このファイルは地球惑星科学専攻の場所のアクセスマップです。この地惑専攻のウェブサーバには皆さんのアカウントはありませんから SCP は使えません。このような時は **wget** コマンドを利用すると良いでしょう。wget は Web からファイルをダウンロードするコマンドで、HTTP・HTTPS・FTP のプロトコルに対応しています。上の例だと、

```
$ wget http://www.eps.s.u-tokyo.ac.jp/access.html
```

のようにして使います。“-x”オプションを用いれば、ディレクトリ構造を保ってダウンロードできます。つまり、

```
$ wget -x http://www.eps.s.u-tokyo.ac.jp/access.html
```

とすると、カレントディレクトリに www.eps.s.u-tokyo.ac.jp というディレクトリができ、その中に access.html が保存されます。

また、“-r”オプションを用いれば、指定したページに含まれるリンク先を再帰的に取得できます。

curl というコマンドでも同じことができますが、詳細については割愛します。

### 3.1.4 練習

`http://www-geoph.eps.s.u-tokyo.ac.jp/~s122621/kadai/index.html` とそのページに書かれているリンク先を全てダウンロードしてください。

## 3.2 テキストフィルター

シェルを使って入力されたデータにフィルターをかけて指定した文字列などを検索したり、変更を加えて出力したいということがあります。ここではそういう時に使用するコマンドの例として検索コマンド grep, find とフィルタリングツールの sed, awk を紹介します。

### 3.2.1 検索 grep

まず検索コマンド grep です。ホームディレクトリで、

```
[1] ~$ ls -a | grep rc
```

```
[2] ~$ grep PATH .bashrc
```

などと打ってみてください。[1] において grep は標準入力（ファイル名リスト）から ‘rc’ を含む行を検索して表示します。[2] は隠しファイル.bashrc 中から ‘PATH’ を含む行を検索して表示します。このように grep は指定したファイルもしくは標準入力から指定された文字列を含む行のみを出力するコマンドです。例えば大量にある Fortran ファイルから module を含む行を表示させたい場合、

```
~$ grep -n module *.f90
```

などとしてファイルを調べることもできます。‘-n’ は行数も表示させるオプションです。

検索繋がりで、ファイルが行方不明になった時などに使えるファイル検索コマンドを紹介します。以下のコマンドは、s192630 さんのホームディレクトリの下にある拡張子が f90 のファイルを検索します。実行してみてください。

```
$ find ~s192630/ -name *.f90
```

### 3.2.2 ストリームエディタ sed

次はフィルタリングツール *sed* (stream editor) です。sed を使うと入力に対して様々なフィルター操作をかけて出力することができます。そのうちのいくつかをここで紹介します。

| コマンド | 説明 | 例 |
|---|---|---|
| s | 文字列の置換 | sed ’s/置換される文字列/置換する文字列/’ |
| p | ある文字列が含まれる行の出力 | sed ’/ある文字列/p’ |
| d | ある文字列が含まれる行の削除 | sed ’/ある文字列/d’ |
| c | ある文字列が含まれる行の置換 | sed ’/ある文字列/c\置換する文字列’ |

このうち特に置換のコマンド s はよく使いますのでもう少し詳しく見ていきましょう。例えば before を after に置換するためには次のようになります。（元のファイルは変化しません）

```
[1] ~$ sed 's/before/after/g' filename
[2] ~$ something | sed 's/before/after/g'
```

ここでは g をつけていますが g をつけると読み込んだ行の中の全ての文字列に対して、無い場合は各行の 1 つ目のみに対して置換を行います。例えば

she sells UMI shells by the UMI shore

の UMI を sea に置換したいとしたら二つ目の UMI を置換するためには g を最後につける必要があります。またコマンドにはオプションをつけることができます。いくつか紹介すると、操作結果に応じて入力ファイルを上書きする-i、二つのコマンドを同時に行うための-e、変更を加えた行のみの出力行う-n のオプションなどがあります。(注：デフォルトでは読み込んだ全ての行が出力されます。) さらに*、、^$、.、などのメタキーを上手に使うことでより細かな操作ができるようにな

ります。メタキーの使い方は各自で調べて見てください。

### 3.2.3 awk

最後に awk です。これは 1 つの言語ですのでおよそありとあらゆることができますが、簡単な使い方を紹介します。基本構文は sed と同じです。

```
[1] ~$ awk '{命令}' filename
[2] ~$ something | awk '{命令}'
```

ファイルの中身または標準入力に対して、1 行ずつ” 内で指定した処理を行います。

例を見ていきましょう。先ほどの ‘ls -l’ でユーザ名とファイル名だけを表示させたい場合、つまり各行の 3 要素めと 8 要素目を表示させるには、

```
~$ ls -l | awk '{print $3,$8}'
```

と打ちます。print は Fortran の write (*,*) や Python の print() に対応します。$3 は 3 要素目という意味です。うまくいきましたか？ 次に先ほどの出力の 1 行目を見てみましょう。空行が出ています。これは入力の 1 行目がファイル容量の合計 ‘total #’ であるため、要素数が 2 つしかないからです。3 要素目と 8 要素目は空だとして空行を出力します。これに対処するには

```
~$ something | awk ' 条件{命令}'
```

という構文を使います。各行に対して条件が当てはまるか検討し、当てはまれば命令を実行します。試しに次の 2 つを実行してみてください。

```
[1] ~$ ls -l | awk 'NR>1{print $3,$9}'
[2] ~$ ls -l | awk 'NF>2{print $3,$9}'
```

のいずれも 1 行目の空行を出力しないでくれると思います。NR は現在の行数、NF は現在の行の要素数を表します。行数が 1 行目より後のとき実行せよ、または、要素数が 2 個より多いとき実行せよという意味です。

## 3.3 四則演算 expr

簡単な整数の計算をするコマンド expr の紹介です。

```
~$ expr 1 + 2
```

のようにして使います。+、-、、/、% がそれぞれ足し算、引き算、かけ算、割算、余りに対応します。空白は必須なので注意してください。

# 第 4 章

# シェルスクリプト

## 4.1　シェルスクリプト始めの一歩

シェルスクリプトとは、シェルで実行するコマンドを一つにまとめたファイルのことです。複数のコマンドを用いるとき、その途中で１箇所でも間違えてしまうと、最初からやり直しということにもなりかねません。そのような手間を軽減するために、行いたい作業を１つのファイルにまとめておき、コマンド１つでファイル内の作業をすべて実行できると便利ですね。gnuplot の plt ファイルに似たものとも考えられます。

### 4.1.1　Hellow World!

ここから配付したサンプルファイルを使っていきます。さっそく sample01.sh を見てみましょう。

Listing 4.1　sample01.sh

```
1 #!/bin/sh
2 echo Hello World!
```

たった 2 行しかありませんがこれも立派なシェルスクリプトです。次のコマンドによって、実行することができます。

```
~$ sh sample01.sh
Hello world!
```

ここで、`ls -l` をして、sample01.sh の詳細情報を見てみましょう。

```
~$ ls -l sample01.sh
-rw-r--r-- 1 s212600 20000 29 Jun  9 13:48 sample01.sh
```

続いてコマンドライン上に次のようにタイプしてみましょう。

```
~$ ./sample01.sh
-bash: ./sample01.sh: Permission denied
```

エラーが出てしまい、実行できませんでしたね。そこで次のコマンドを実行してみてください。

```
~$ chmod +x sample01.sh
```

`chmod` は、**ファイルの属性を変えるコマンド**でしたね。もう一度、詳細情報を見てみましょう。

```
~$ ls -l sample01.sh
-rwxr-xr-x 1 s212600 20000 29 Jun  9 13:48 sample01.sh
```

などと表示され、ファイルの属性が変わっていることがわかりますね。(各権限に x が加わりました)。chmod に +x というオプションをつけることで、sample01.sh というファイルに**実行権**を与えることができます。

```
~$ ./sample01.sh
Hello world!
```

先ほど sh sample01.sh としたときと同じ結果が得られたのではないかと思います。実行権を与えるというのは、**それ単独で動くようにする** (sh コマンドは必要ない) ということです。[*1]。

sample01.sh に 1 行付け加えてみましょう。

Listing 4.2　sample01.sh 改

```
1  #!/bin/sh
2  echo Hello World!
3  pwd
```

```
~$ ./sample01.sh
Hello World!
(今いるディレクトリの絶対パス)
```

となるはずです。追加された 3 行目に対応する出力は、普通にターミナルから pwd と入力したときと同じものです。

以上のことから、**シェルスクリプトは、単にコマンドを並べればよい**ということがわかりますね。

### 4.1.2　1 行目について

シェルスクリプトの 1 行目の

```
1  #!/bin/sh
```

は、”Shebang”（シバン/シェバン）と呼ばれるものです。

ファイルの 1 行目の行頭が#!の場合は、そのファイルの実行をカーネル (Unix の中核) に要求します。カーネルは#!のあとの絶対パスで指定されるファイル名を見て、そのファイルをプログラムとして起動します。そして、2 行目以降に書いてある内容をそのプログラムの入力とします。つまり、このルールからすればシェルスクリプトに限定された書き方ではなく、例えば

```
1  #!/usr/bin/gnuplot -persist
2  plot sin(x)
```

[*1] Fortran で、gfortran でコンパイルした後に実行ファイルに対して chmod で実行権を与えなくても
$ ./a.out
で実行できるのは、すでに gfortran によって a.out に実行権が与えられているからです。

というファイルを作って実行したときには、gnuplot で描かれた $\sin x$ のグラフが現れるわけです[*2]。(`persist` は gnuplot の、グラフを画面に残しておくというオプションです。)

**練習問題 1**

(1) sample01.sh(改) の 4 行目以降に、好きなコマンドを入力・保存して、実行してみましょう。

(2) さらに、次の 2 行を加えて保存・実行してみてください。

```
cd ..
pwd
```

全部の実行が終わった後でもう一度ターミナルに `pwd` と入力し、自分がいまどこにいるのかを確かめてください。どんなことがわかりますか？

[*2] 筆者の環境では、gnuplot は/usr/bin/gnuplot にあるそうなので、これで通りました。bad interpreter: No such file or directory というエラーが出るときには、gnuplot のパスが異なる可能性があります。which gnuplot で gnuplot のパスを調べて、適宜書き換えてください。

## 4.2　シェルの基本機能とシェルスクリプト

先ほど勉強したシェルの基本機能をシェルスクリプトでも使ってみましょう。

### 4.2.1　シェル変数

Listing 4.3　sample02.sh

```
1  #!/bin/sh
2  TORI1=inko
3  TORI2=fukuro
4
5  echo $TORI1
6  echo $TORI2
7  echo $TORI3
8  echo naku$TORI1
9  echo $TORI1naku
10 echo naku${TORI2}
11 echo ${TORI2}naku
```

実行権を与え、実行してみましょう。

```
~$ ./sample02.sh
inko
fukuro

nakuinko

nakufukuro
fukuronaku
```

実行結果は理解できるでしょうか？

`$TORI3` は未定義の変数です。Fortran では、`implicit none` と書いておけば未定義の変数があっても gfortran に怒られてコンパイルエラーになりました。しかしシェルスクリプトでは未定義の変数は null 値で勝手に初期化され、エラーにはなりません。従って、シェルスクリプトでは変数に注意してコーティングする必要があります。また`$TORI1naku` も未定義の変数とみなされました。このような場合、変数名を明示的に`${ }`で囲む必要があります。

次のサンプルプログラムです。

Listing 4.4　sample03.sh

```
1  #!/bin/sh
2  name1=`whoami`
3  name2=$(whoami)
4  echo ${name1}
```

```
5 echo ${name2}
```

実行してみると、あなたのユーザ名が標準出力に出てきたはずです。

バッククォート‘ ‘(日本語キーボードでは Shift+@、US キーボードでは 1 キーの左隣) または$( ) を用いて、変数にコマンドの実行結果を代入することができました。

sample03.sh を次のように書き換えてみましょう。

Listing 4.5 sample03.sh 改

```
1 #!/bin/sh
2 name1=`echo I am \`whoami\`.`
3 name2=$(echo I am $(whoami).)
4 echo ${name1}
5 echo ${name2}
```

実行すると、例えばユーザ名が s212600 のひとは、

```
~$ ./sample03.sh
I am s212600.
I am s212600.
```

と返ってくるはずです。2 行目で、‘の前にエスケープシーケンスがついているのは、メタキャラクタと区別するためです。3 行目の書き方ではエスケープシーケンスは必要ありませんね。

ssh 先にリダイレクトすることもできます。先ほど改めた sample03.sh を dover に投げる場合は、このようにします。

```
~$ ssh dover < sample03.sh
I am s212600.
I am s212600.
```

このように返ってくるはずです。

さらに次のように書き換え、sample04.sh とします。

Listing 4.6 sample04.sh

```
1 #!/bin/sh
2 name1=`echo I am \`whoami\`. | rev`
3 name2=$(echo I am $(whoami). | rev)
4 filename=revname.txt
5 echo ${name1} > ${filename}
6 echo ${name2} >> ${filename}
```

**練習問題 2**

sample04.sh を作成・実行して、出力ファイルの内容を確認してください。

### 4.2.2 特殊変数

シェルスクリプトには、特殊変数と呼ばれる特別な変数があります。

| 特殊変数 | 説明 |
|---|---|
| $0 | シェルスクリプトの名前 |
| $1,$2,...$9 | シェルスクリプトの 1 番目,2 番目...9 番目の引数 |
| $# | シェルスクリプトの引数の数 |
| $* | シェルスクリプトの引数全部 |
| $@ | シェルスクリプトの引数全部 |

表 4.1: シェルスクリプトの特殊変数

特殊変数を用いて、引数をとることができます。expr は整数の四則演算をするコマンドです。

Listing 4.7　sample05.sh

```
1 #!/bin/sh
2 echo result: `expr $1 + $2`
```

```
~$ ./sample05.sh 5 10
15
```

のように実行すると上手く動きます。引数の値は自由に変えてみてください。また、2 行目を echo `expr $1+$2*$3`などとしていろいろ遊んでみてください。ちなみに$3 まで使えば、当然引数は 3 個必要です。

ところで、シェル変数には型の概念がないことは前に説明しました。5 や 10 のような数は整数として適宜判断してくれています。ためしに同じ sample05.sh を

```
~$ ./sample05.sh carrot eggplant
```

のように実行してみるとどうなりましたか？

### 4.2.3　ヒア・ドキュメント

またまたサンプルファイルです。

Listing 4.8　sample06.sh

```
1 #!/bin/sh
2 cat <<EOF
3 set terminal pdfcairo
4 set output 'sin.pdf'
5 plot sin(x)
6 EOF
```

```
~$ ./sample06.sh
set terminal pdfcairo
set output 'sin.pdf'
plot sin(x)
```

これは 2 行目の<<EOF から 6 行目の EOF の間に書かれている内容がそのまま cat の標準入力に渡されて、cat の標準出力にこれが出力されたというわけです。このような機能をヒア・ドキュメントといいます。

これをみているといかにも gnuplot のコマンドですね。もう少し解説します。シェルは<<というマークを見つけると、それに続く文字列 (今の場合は EOF) を覚えて、次に同じ文字列が出てくる直前の行までをその前のコマンドの標準入力に渡します。実は EOF でなくても、orz でも SHIROHARAINKO でも何でも良いわけです (記号文字は不可)。EOF というのは “End of File” の略で、一般的には EOF が使われます。(渡したい文字列に”EOF” が含まれる場合は、当然 EOF 以外にする必要があります。)

もう一つ大事なこととして、ヒア・ドキュメントの中では変数が展開されます。つまり、sample06.sh を少し書き換えて、このようにしてみましょう。

Listing 4.9　sample06.sh 改

```
1  #!/bin/sh
2  number=1
3  cat <<EOF
4  set terminal pdfcairo
5  set output 'sin${number}.pdf'
6  plot sin(${number}*x)
7  EOF
```

シェル変数 number の部分が展開されて、

```
~$ ./sample06.sh
set terminal pdfcairo
set output 'sin1.pdf'
plot sin(1*x)
```

と返ってきます。

さていよいよ、cat の部分を gnunplot に変えて実行してみましょう。ヒア・ドキュメントの中身がそっくりそのまま gnuplot に渡されて、$\sin x$ のグラフが sin1.pdf というファイルに書かれたはずです。

ここまでくれば、シェルスクリプトもかなり便利なものになってきたのではないでしょうか？

## 4.3　シェルスクリプトの制御文

### 4.3.1　ループ制御：for 文

Fortran の do ループのようなこともできます。ただし、表式はだいぶ違います。例えば指定された回数だけコマンドを実行するような場合、for 文が使えます。構文は次のような感じです。

```
1  for 変数 in 引数1 引数2 引数3 ...
2  do
3      コマンド
4  done
```

サンプルも使ってみましょう。

Listing 4.10　sample07.sh

```
1 #!/bin/sh
2 for i in 1 2 3 4 5
3 do
4     echo ${i}
5 done
```

実行してみると、、、

```
    ~$ ./sample07.sh
    1
    2
    3
    4
    5
```

となりましたか？ 要するに、2 行目の in の後ろに並んだ引数が順番に変数の部分に入って、あとは do と done で囲まれた部分が、引数が空になるまで繰り返されるということです。

別に in の後ろは数字でなくても構いません。この辺は Fortran より自由な感じですね。例えば、

Listing 4.11　sample07.sh 改

```
1     #!/bin/sh
2     for i in sekiseiinko kozakurainko okameinko
3     do
4     echo ${i}
5     done
```

などと書いておけば、たくさんの鳥を出力することができます。

話を戻して、in のあとに先ほど学んだバッククォートによるコマンド置換を使ってみることにします。ターミナルで

```
        ~$ seq 1 1 5
```

としてみてください。

```
    1
    2
    3
    4
    5
```

と返ってくるはずです。ということは先ほどの sample07.sh の 2 行目の in のあとを

Listing 4.12　sample07.sh 改 2

```
1     #!/bin/sh
2     for i in `seq 1 1 5`
```

```
3     do
4     echo ${i}
5     done
```

などとすれば、in のあとに 1 2 3 4 5 と書いたのと同じことになるわけです。引数が 5 つくらいであれば別にありがたみを感じませんが、1 から 100 までとかになれば効果覿面ですよ。

また bash では、C 言語風に次のように書くこともできます。

Listing 4.13　sample07.sh 改 3

```
1     #!/bin/sh
2     for (( i=1 ; i <= 5 ; i++ ))
3     do
4     echo ${i}
5     done
```

**練習問題 3**

sample06.sh と sample07.sh を参考にして、$\sin(x), \sin(2x), ... \sin(5x)$ のグラフをそれぞれ sin1x.pdf, sin2x.pdf, ... sin5x.pdf に出力するシェルスクリプト sample08.sh を作ってください。

### 4.3.2　ループ制御：while 文

for 文の他に、while 文によるループの書き方もあります。while の後の条件式が真である間、実行され続けます。

Listing 4.14　sample07.sh 改 4

```
1     #!/bin/sh
2     i=1
3     while [ ${i} -le 5 ]
4     do
5     echo ${i}
6     i=$(expr ${i} + 1)
7     done
```

無限ループも作れます。break でループを抜けます。(if 文は次節を参照)

Listing 4.15　sample07.sh 改 5

```
1     #!/bin/sh
2     i=1
3     while true
4     do
5     echo ${i}
6     if [ ${i} -eq 5 ]
7     then
8         break
```

```
9     fi
10    i=$(expr ${i} + 1)
11    done
```

### 4.3.3 条件分岐：if 文

シェルスクリプトでも if 文が使えます。使い方は次のとおりです。

```
1     if 条件式1
2     then
3     条件式1が真のときに実行される命令
4     elif 条件式2
5     then
6     条件式1が偽で条件式2が真のときに実行される命令
7     elif 条件式3
8     then
9     条件式1と条件式2が偽で条件式3が真のときに実行される命令
10    ...
11    else
12    すべての条件式が偽のときに実行される命令
13    fi
```

if が fi で終わるあたりに遊び心を感じますが、Fortran に似ていますね。elif なのにも注意が必要です。よく間違えます。条件式の部分は大カッコ + 空白 ( [␣) と、空白 + 大カッコ ( ␣] ) でくくる必要があり、表 8 に示すようなものがあります。始めのカッコの直後と終わりのカッコの直前に空白が必要なことに注意してください。

| 条件式 | 真になる条件 |
|---|---|
| [ 文字列 1 = 文字列 2 ] | 文字列 1 と文字列 2 が同じ |
| [ 文字列 1 != 文字列 2 ] | 文字列 1 と文字列 2 が同じでない |
| [ -n 文字列 ] | 文字列が空でない (not zero) |
| [ -z 文字列 ] | 文字列が空 (zero) |
| [ 整数 1 -eq 整数 2 ] | 整数 1 と整数 2 は等しい (equal) |
| [ 整数 1 -ge 整数 2 ] | 整数 1 は整数 2 以上 (greater equal) |
| [ 整数 1 -gt 整数 2 ] | 整数 1 は整数 2 より大きい (greater than) |
| [ 整数 1 -le 整数 2 ] | 整数 1 は整数 2 以下 (less equal) |
| [ 整数 1 -lt 整数 2 ] | 整数 1 は整数より小さい (less than) |
| [ 整数 1 -ne 整数 2 ] | 整数 1 と整数 2 は等しくない (not equal) |
| [ -d ファイル名 ] | ファイルはディレクトリ |
| [ -f ファイル名 ] | ファイルは通常ファイル |
| [ -r ファイル名 ] | ファイルは読み出し可能 |
| [ -s ファイル名 ] | ファイルは長さが 0 バイトでない |

| | |
|---|---|
| [ -w ファイル名 ] | ファイルは書き込み可能 |
| [ -x ファイル名 ] | ファイルは実行可能 |
| !(条件式) | 直後に続く条件式の結果を否定 |
| [[ (変数) =~ (正規表現) ]] | 変数が正規表現を満たす |
| [ (1つ目の条件式) ] && [ (2つ目の条件式) ] | 2つの条件式の論理積（and/かつ） |
| [ (1つ目の条件式) ] \|\| [ (2つ目の条件式) ] | 2つの条件式の論理和（or/または） |

表 4.2: 条件式のあれこれ

例えば、このようなものがあります。

Listing 4.16　sample09.sh

```
1  #!/bin/sh
2  your_gid=`id -g`
3  if [ $your_gid -eq 20000 ]
4  then
5      echo 'You are a student.'
6  elif [ $your_gid -eq 20090 ]
7  then
8      echo 'You are a TA.'
9  else
10     : # do nothing
11 fi
```

2 行目の <id -g> で自分のグループ ID を得ることができます。dover では学生 (グループ名 student) には 20000、TA(グループ名 ta) には 20090 という ID を与えています。このシェルスクリプトを dover に投げます。

```
~$ ssh dover < sample09.sh
```

このスクリプトを皆さんが実行すると 3 行目の評価式が真になって 5 行目が実行され、

```
You are a student.
```

と表示されるはずです。ちなみに:は何もしないというコマンドです。

ほかにも case 文や until 文などがあります。気になる人は調べてみてください。

## 4.4　その他

### 4.4.1　コメント

Fortran で行頭に!と書くと行全体がコメントになったのと同じように、シェルスクリプトでも#と書くと、この文字以降の 1 行がコメントとみなされます。行頭に#があればその行全体がコメント文になります。また、何も書かない空白行が

あった場合、これも無視します。

また、複数行のコメントアウトをするときは、

```
1    #!/bin/sh
2    <<COMMENT
3    （プログラム）
4    COMMENT
```

というように、<<と任意の文字列 (ここでは COMMENT) を用いると、その間の行がコメントアウトされます。これは、ヒア・ドキュメントの機能によく似ていますね。

以上のことをうまく組み合わせて、読みやすいスクリプトを書きましょう。

### 4.4.2 ;（セミコロン）

一つのコマンドは、改行またはセミコロンで終結します。改行は良いとして、セミコロンで終結するというのは、単に 1 行に複数のコマンドを書けるというだけのことで、さきの sample07.sh を

Listing 4.17 sample07.sh 改 6

```
1    #!/bin/sh
2    for i in `seq 1 1 5`
3    do
4    echo ${i}
5    done
```

と書いても、

Listing 4.18 sample07.sh 改 7

```
1    #!/bin/sh
2    for i in `seq 1 1 5` ; do
3    echo ${i}
4    done
```

と書いても同じということです。Fortran の do ループのように見せかけたいというので使う人もいます。同じことが、if 文の then や、while 文の do にも使えます。好みの問題なので、使わなくても構いません。

# 第 5 章

# まとめ

色々なことをしゃべりましたが、UNIX の仕組みについて重要な点をいくつかまとめておきます。

- コマンドはパスを通すことで、便利に使うことができている。
- パスは設定ファイルの中で定義されている。
- UNIX には様々な設定ファイルがあり、それらの構造を理解し、書き方を学ぶことでいろいろなカスタマイズができる。

UNIX と仲良くなれるような気になってきましたか？ コマンドラインの操作も起動などの仕組みも、実際には慣れによる部分が大きいのですが、ちゃんと順番を追っていけば、それほど意味不明なものではありません。

Windows などの普及している OS では、中で何が起こっているか、使っているだけではほとんど分かりません。逆に UNIX では、中で何が起こっているのかを理解しなくては、かなり意味不明なシステムになってしまいます。そのため、Windows などに慣れているユーザにとって、UNIX は難しい OS だと思われるのでしょう。しかし、中で起こっていることが分かっていれば、かゆい所に手が届くカスタマイズができるわけです。もちろん、それには相応の知識が必要になるわけですが、今日の講義でおぼろげな概観をつかんでもらえたなら、とても嬉しいです。

# 第 6 章

# 課題

今回の講義とテキストの内容を踏まえて、いくつか課題を出します。締切は次の通りです。

**課題 1-?**　4 月 13 日 (水) 23:59 JST (1 週間後)
**課題 2-1**　6 月 1 日 (水) 23:59 JST
**課題 3-?**　6 月 29 日 (水) 23:59 JST

4 月 7 日の時点では難しい課題もあると思いますが、これから様々な課題をこなすうちにわかるようになってくるとおもいます。

## 提出方法

559 室の sakura または edu 上`/home2/mori2022/TA2022/`ディレクトリに、`s2226??/`という名前 (`s2226??`は自分のユーザ名) でディレクトリを作り、そこに各課題の解答ファイル (計 3 つ) を置いてください。全てについて、パーミッションは適切に設定してください[*1]。

## 6.1　課題 1-1: パイプ・リダイレクト

`~mori2021/exercise` ディレクトリの中に、`earthquake.txt` というファイルがあります。このファイルには 1949 年から 2016 年までの、震度別の地震の回数のデータが入っています[*2]。このデータの中から 1965 年から 2011 年までの期間、震度 1 および震度 7 のものを抜き出してください。結果は `kadai1.txt` に出力してください。

このファイルは半角スペースではなく、タブを使って列が区切られていることに注意しましょう。講義で扱っていないコマンドを使用しても構いません。

## 6.2　課題 1-2: PATH の設定

`echo-sd` という文字を装飾して表示するコマンドがあります[*3]。例えば、

[*1] 友達がカンニングできないように、また TA が採点できるように。他人がファイルを書き換えるのも防ぎましょう。
[*2] http://www.data.jma.go.jp/svd/eqdb/data/shindo/index.php
[*3] http://github.com/fumiyas/home-commands/blob/master/echo-sd

~$ echo-sd ’（ここに文字列を入力）’

とすると文字列が豪華になって返されると思います。といいましたが、パスが通っていないと実行できません。/home2/mori2021/TA2021/に echo-sd の実行ファイルがあるので、ここに PATH を設定して実行できるようにしてください。その後、適当な文字列を表示させて結果を kadai2.txt に記入してください。

## 6.3 課題 2-1: sed の練習

/home2/hosotani2021/に sed_kadai.txt があります。sed_kadai.txt はルイスキャロルの “The Walrus and the Carpenter” の文に変更を加えたものです。sed_kadai.txt に対して以下の操作を sed を使って行い、sed_(自分の学籍番号).txt と sed_explanation_(自分の学籍番号).txt を作成し提出してください。

1. 原文で Oyster の部分を全て AWABI に変更してあります。これを元の Oyster に戻してください。
2. 文の途中に

   HOTATE HAMAGURI SAZAE TSUBU KAKI SHIJIMI

   という行を何行か加えてあります。これらの行を取り除いてください。
3. 行頭の the のみ The で置き換えてください。(注:行の先頭の there や行の途中の the まで The に置き換えないこと。)

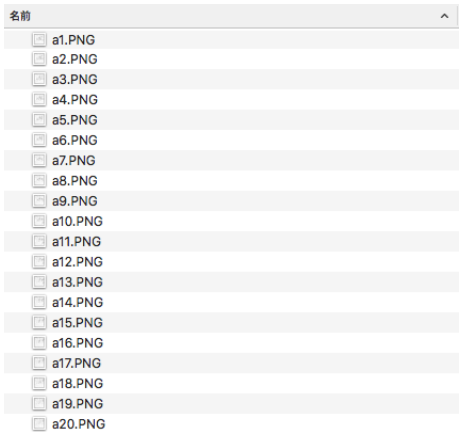
この 3 処理をした文を sed_(自分の学籍番号).txt に出力して、どのように処理をしたか sed_explanation_(自分の学籍番号).txt に説明してください。

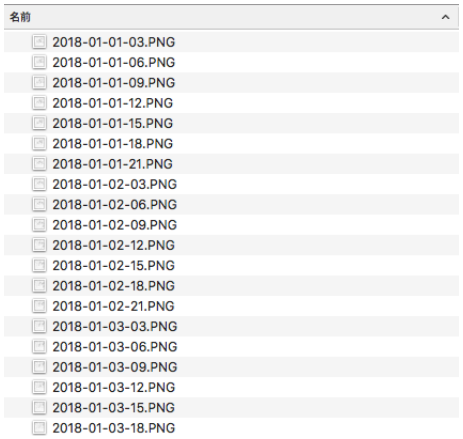
## 6.4 課題 3-1：ファイル整理

気象庁は毎日 3,6,9,12,15,18,21 時 (0 時はありません) に速報天気図を発表しています。これを月ごとにまとめたフォルダを 4 つ (フォルダ名が 2016_02,2017_02,2017_11,2018_01) 用意しました。それぞれのフォルダには, 2016 年 2 月, 2017 年 2 月, 2017 年 11 月, 2018 年 1 月の天気図が, 各月の 1 日 3 時のものから最終日の 21 時のものまで連番 (a1.PNG,a2.PNG,...) で並んでいます (ただし欠番もあります！)。これらの連番のファイル名を,「20××-××-××-××.PNG」(年 4 桁-月 2 桁-日 2 桁-時 2 桁。月日と時刻については，2 桁でない場合にはゼロ埋めする。) という形式に変更し，一目でそのファイルがいつの天気図であるかわかるようにできるシェルスクリプトを作成してください。(下の図の左側のように並んでいるものを，右側のように変更するということです)。ただし，シェルスクリプトを実行する際には，年と月の 2 つの引数を必要とするもの (シェルスクリプトを shell2 ディレクトリ直下に置き，<./××.sh 2016 2> のように実行する) にしてください。コマンドを実行する際の引数の形式（2 月を指定するのに 2 とするのか 02 とするのかなど）についての解説をコメントアウトという形でスクリプト内に記述しておいてください。ファイルに欠番があるときには，何年何月何日何時の天気図が存在しないのか標準出力で表示してください。また，各月の日数，閏年に注意して必要に応じて場合分けを行いましょう。採点には，他の年月のファイルで試します。

以下，やり方に見当もつかない人のためのヒントです。

- 31 日まである月は 1,3,5,7,8,10,12 月で，30 日までしかない月は 4,6,9,11 月です。
- 2 月は 4 で割り切れる年は 29 日まで，それ以外の年は 28 日までです。ただし，100 で割り切れる年は 28 日までです。ただし，400 で割り切れる年は 29 日までです。




- 1 日あたり 7 枚発行されるので，順に並べた時，3 時はすべて 7 で割って 1 余る順番目，6 時はすべて 7 で割って 2 余る順番目ですね。

## 6.5　課題 3-2：マシンスペック

あなたはいま使用している UNIX マシンのスペックを知りたくなったとします。Windows では “Win キー +Pause キー” や “msinfo32.exe”、Mac では “この Mac について” やターミナルで “system_profiler SPHardwareDataType” などとすることでコンピュータのスペックを知ることができますが、UNIX の CUI にはそのような便利なコマンドはありません。そこで、以下のような仕様を満たす、実行すると UNIX マシンのスペックを表示するシェルスクリプトを作りなさい。

```
~$ bash spec.sh
Hostname : ホスト名+ドメイン名（ホストのIPv4アドレス）
OS       : OSの名前
CPU      : CPUの名前
number of physycal CPUs : 物理CPU数
number of cores / CPU   : CPU1個あたりのコア数
number of threads / CPU : CPU1個あたりのスレッド数
number of total threads : 全体のスレッド数
Memory   : メモリ容量
```

dover での実行例です。

```
~$ bash spec.sh
Hostname : dover.eps.s.u-tokyo.ac.jp  (133.11.229.15)
OS       : Debian GNU/Linux 8
CPU      :  Intel(R) Celeron(R) CPU E3400 @ 2.60GHz
number of physycal CPUs : 1
number of cores / CPU   : 2
number of threads / CPU : 2
number of total threads : 2
```

```
Memory    : 2036668 KB (1.94 GB)
```

以下のコマンドを利用してください。

- ホスト名 + ドメイン名は `hostname -A` を実行すると表示されます。[*4]
- ホストの IPv4 アドレスは `hostname -I` を実行すると表示されます。
- OS の名前は `/etc/issue.net` に書いてあります。
- CPU の情報は `/proc/cpuinfo` に書いてあります。ただし、各 CPU スレッドについての情報が段落ごとにリストアップされています。`lscpu` というコマンドでも CPU に関する情報を得ることができます。
  - CPU の名前は、最初の段落の “model name” を見ればよさそう。
  - 物理 CPU 数は、“phisycal id”(0,1,2...) が何種類あるか数えればよさそう。
  - CPU1 個あたりのコア数は、最初の段落の “cpu cores” を見ればよさそう。
  - CPU1 個あたりのスレッド数は、最初の段落の”“sibling” を見ればよさそう。
  - 全体のスレッド数は、段落の数を数えればよさそう。または計算すればよさそう。
- メモリの情報は `/proc/meminfo` に書いてあります。`lsmem` というコマンドでも見ることができます。
  - メモリ容量は、“MemTotal” を見ればよさそう。
  - でも KB 単位でわかりにくいので、GB 単位も追記してくれたら加点します。
- 使えそうなコマンド：`grep -m, awk, sed, sort -u, wc -l, tr -d ' '`(空白除去), `bc...`

もちろんここにないコマンドや方法を使っても実行できれば大丈夫です。[*5]

シェルスクリプトができたら、まず自分の PC（Ubuntu 仮想環境）で実行した結果を確認してください。

次に、dover に scp で持って行って、上と同じ結果になることを確認してください。

そして、asano に持って行って、asano のスペックを確認してみましょう。自分の PC や dover と比べてみて、どうですか。asano での実行結果を”s2226**_asano.txt” にリダイレクトし、シェルスクリプト”s2226**_kadai2.sh” とともに提出してください。

提出されたシェルスクリプトは、計算機演習のマシン (dover,asano など) 以外のマシンでもうまく実行できるかテストします。

## 6.6 課題 3-3：論文の共著者

科学論文は研究者が単独で執筆することは少なく、共著者がいることが多いです。大きなプロジェクトに関係する論文だと著者が数十人にもなることもあります。では今まで出版された論文で、最も著者が多い論文では著者は何人ほどいるでしょうか。それは、“Aad, Georges, et al. “Combined Measurement of the Higgs Boson Mass in p p Collisions at $\sqrt{s} = 7$ and 8 TeV with the ATLAS and CMS Experiments.” *Physical review letters* 114.19 (2015): 191803.” という論文で、なんと 5,154 人もの著者がいます。

この論文の著者のリストが “authors.txt” です。3 行目に著者の一覧が 1 行で書かれています。それぞれの著者名には研究機関の番号が振られていて、5 行目以降に研究機関の番号と研究機関名、そして国名が書かれています。

[*4] ディストリビューションによっては `hostname -f` です。ディストリビューションの違いによって実行コマンドを変える実装はかなり大変で、この課題で求めているレベルを超えるので、Debian(dover や asano) で動くシェルスクリプトを作ってもらえれば大丈夫です。

[*5] インターネットで調べるとヒントがたくさんでてくるかもしれません。インターネットを使って積極的に勉強し、いままでの授業で出てきていないコマンドを使うのは全く問題ありませんが、その場合はそのコマンドの簡単な説明をコメントアウトで入れておいてください。（要するに、コマンドの意味がわからないままコピペしたということのないようにしてください）資料のコマンド一覧に載っているものについては説明はいりません。

さて、この中に例えば「日本の研究者」は何人いるでしょうか。ここで言う「日本の研究者」とは「日本の研究機関に属する研究者」を意味することにします。

例えばこのようにして調べられそうです。

- ”Japan” の研究機関を検索し、その研究機関番号をリストアップする。ただし「4a」「4b」「4c」のような番号もあるので注意が必要です。
- その研究機関番号がつけられた研究者をリストアップする。
- 研究者のリストを並べ替え、重複があれば削除し、カウントする。

以上を踏まえて、authors.txt から「国名を与えると、その国の研究者をアルファベット順にリストアップし、人数を表示する」シェルスクリプトを作りなさい。

## 6.7 採点基準

採点は次の規準に則り 4 段階で評価します。

期限内に,

A 評価　意図された課題ができた上で，閏年の例外や引数への配慮（課題 3-1），メモリ容量の換算（課題 3-2）などさらに発展させた課題を提出した場合。

B 評価　意図された課題ができた場合。

C 評価　意図された課題にまではたどり着かなかった場合でも，十分に思考した痕跡のある課題を提出した場合。

D 評価　いずれの課題も提出しなかった場合，もしくは思考の痕跡がほとんど見られない課題を提出した場合。

評価に応じて得点が割り当てられます。期限内に課題を提出したものの，パーミッションが不適切で TA が開けない場合，後日連絡します。その場合，開ける課題を再提出してもらえれば，上記規準から 1 段階下げた評価を与えます。

# 第 7 章

# 付録

## 7.1 Linux と SSH と GUI

みなさんはこれまで Linux がインストールされた asano というコンピュータに SSH で接続してさまざまな課題を解いてきました。このとき、みなさんはずっとターミナル上で文字のみで操作する CUI を用いてきました。しかし、Linux は CUI でしか操作できないわけではありません。

### 7.1.1 X Window System

Linux で GUI を実現するのに欠かせないソフトウェアが X Window System です。X Window System は、「こういった画面を表示しなさい」という司令を出す「X クライアント」と「ではディスプレイのピクセル?番は?色ですね」と実際に表示をする「X サーバ」の 2 つの部分からなります。この仕組によって、画面全体の各ピクセルの色という莫大なデータを送らずとも GUI が表示できるようになっています。そして、この「司令」は SSH 経由でも送信することができるのです。

### 7.1.2 SSH で GUI ソフトを起動する

通信を暗号化した遠隔ログイン方法として広く使われているのが SSH (Secure SHell) です。やり方は以下の通りです。
SSH を利用してリモートホスト (遠隔地にあるサーバ) にログインするには、

```
$ ssh username@hostname
```

とします。ホスト名の部分には IP アドレスをそのまま書いても構いませんが、普通はドメイン名を入力します。例えば、

```
$ ssh s2126??@dover.eps.s.u-tokyo.ac.jp
```

といった具合です。入力後、パスワードを聞かれたら入力してリターンしてください。
なお、ローカルホスト (本人の手元にある端末) とリモートホストとでユーザ名が一致している場合は、

```
$ ssh hostname
```

と username を省略してもＯＫです。
初めて ssh でログインするホストにアクセスしたときは、

```
The authenticity host 'edu 05 (192.168.1.105)' can't be established.
DSA key fingerprint is cb:27:ec:3c:ff:02:f6:fc:9e:7b:39:80:e7:0f:9e:bf.
Are you sure you want to continue connecting (yes/no) ?
```

などとたずねられますので、

```
yes
```

と打ってください。

リモートホストで作業が終わってログアウトする際には、

```
$ exit
```

と打てばもとのローカルホストでの作業に戻れます。

ここで大事なことですが、ssh の後に「-X」(大文字であることに注意！) というオプションをつけないと、リモートホストでウィンドウを開く系の作業ができません。このオプションをつけることで、X window system (ウィンドウを開いてくれるシステム) も自動的に転送されるようになり、リモートで開いたウィンドウがローカルで見えて操作できるようになるのです[*1]。試しに「-X」なしでログインしてから、

```
$ emacs &
```

と入力して Emacs を起動しようとしてみてください。新しいウィンドウを開けずに、エラーとなってしまうはずです[*2]。その他のアプリケーションについても、ウィンドウを開くことができないはずです。

### 7.1.3 自宅から SSH をして GUI ソフトを起動するには

自宅のパソコンからでも、インターネットに接続されていれば、今みなさんがやっているのと同じように、演習室にリモートログインして作業ができます。これは非常に便利です。Windows にも標準で OpenSSH がバンドルされるようになりましたので、いずれの OS もそれぞれ適当なターミナルからそのまま SSH ができます。ただし、GUI なソフトを使用したい場合は X サーバーというソフトウェアをローカルホストにインストールする必要がありますので、使用している OS に合わせて適切な環境設定をしましょう。

#### Windows の場合

Windows の場合、X サーバーには無料の VcXSrv や有償の X410 などがあります。インストール方法については初回の環境構築で用いた wiki を参考にしてください。

#### Mac の場合

Mac の場合、XQuartz という X サーバーがあります。559 室の edu には標準でインストール済みです。

なお Mac から edu にログインした場合に、日本語が文字化けすることがあります。ログイン後

[*1] たまに**-X** をつけてもうまくいかない場合があります。その場合は**-Y** とすると解決することがあります。

[*2] **$ emacs -nw &**のように **nw** オプションをつけると、現在のシェル上に Emacs が起動するためエラーにはなりません。

```
$ export LANG=ja_JP.UTF-8
```

とすることで解決するかもしれません。

### 7.1.4 .xsession

本小節は旧 *edu* に準拠した記述です。

皆さんが今使っている Linux は、操作こそ CUI ですが、ログインの仕方やログイン後の画面などは Windows や Mac OS X などとそれほど違いを感じられなかったのではないかと思います。元祖の UNIX は完全にコマンドラインの操作のみで、画面全体にターミナルが表示されているような状態なのですが、皆さんが使っているのは、グラフィカルログインという形式で、違和感なくログインでき、マウスによる操作もできますし、Windows のようにウィンドウがぽんぽんと現れてくれます。これを実現しているのが X Window System と呼ばれるものです。単に 'X (エックス)' と呼ばれることもあります。

試しに、`Alt + Control + F3` を同時押ししてみましょう。画面が変わったのではないかと思います。それが X を使っていない状態です。元に戻すには、`Alt + Control + F7` とします。また、テキストログインした状態からでも `startx` とすることで X を起動することが可能です。

ここでは X について細かく説明することはしませんが、`.xsession` というのはこの X Window System に関する設定ファイルです。X が起動する際に実行されます。`.xsession` の他にも、x と付いているファイルは X Window System 用のファイルであることが多いと思います。興味のある人は自分で調べてみましょう。

それでは、`.xsession` の中をのぞいてみましょう。`cat -n .xsession | lv`などとして、指定する行の辺りをながめてみてください。コメントが付いていますので、参考にしてください。

いろいろとありますが、例えば 17 行目の applications という部分は、記述することで X が起動したときにアプリケーションを立ち上げることができます。今は全ての行にコメントアウトの記号 '#' が付いていますので、何も立ち上がらないはずです。その下の Console window や Terms and Editors も似たような感じです。42 行目の wallpaper は文字通り壁紙に関しての記述で、68 行目の Screensaver がスクリーンセーバーに関する記述です。

79 行目からの Window manager という部分は、X のユーザインターフェイスや見た目を選ぶことができる部分と考えれば良いでしょう。皆さんの使っているのは fluxbox と呼ばれるウィンドウマネージャです。世の中にはたくさんのウィンドウマネージャが存在し、それこそ Windows や Macintosh の画面と見間違えるような UI (ユーザインターフェイス) から、コマンドラインでの操作に特化したようなシンプルなデザインもあります[*3]。カスタマイズも GUI でできるものもあれば、エディタでしかできないものもあります。動作の重さもそれぞれです。

自分に合ったウィンドウマネージャを選べば良いのですが、そこに書いてあるすべてのウィンドウマネージャが 210 にインストールされているのかどうかはよく分かりません。ネットなどを調べていて、もし使ってみたいものが見つかったら、admin に相談してみることをお奨めします。

## 7.2 パッケージマネージャ

UNIX にソフトウェアを追加でインストールする方法には複数ありますが、最も簡単なものはパッケージマネージャを使うものです。eduvm には標準で apt(Advanced Package Tool) がインストールされており、Mac では Homebrew が事実上の標準パッケージマネージャです。また、Python では pip というパッケージマネージャが使われており、Anaconda

[*3] fluxbox はどちらかというとシンプルな部類に属します。

を使って Python をインストールした場合は conda という pip より高機能なパッケージマネージャもインストールされます。

他にも様々なパッケージマネージャが存在しますが、それらに共通する機能は次の通りです。

- インストールされているパッケージを把握する。
- オンラインのサーバーと、パッケージの情報を同期する。
- インストールされているパッケージをアップデートする。
- 新しいパッケージをダウンロードし適切なディレクトリに配置する。
- パッケージ間の依存関係を解消する。

これらの機能により、我々はほぼ自動で新しいプログラムをインストールし、かんたんにシステムを最新に保つことができます。

### 7.2.1 apt

前述の通り、eduvm には apt というパッケージマネージャが標準でインストールされています。自分のパソコンが Windows で、WSL 上に Ubuntu をインストールしている場合も同様です。これを使って新しいソフトをインストールするには次の通りの手順を踏みます。

まず、

```
$ sudo apt update
```

として、パソコンが持っているパッケージの情報をオンラインの最新のものへと更新します。このときパスワードを要求されますが、入力するのは今ログインしているユーザーのパスワードです。(eduvm のデフォルトなら、ユーザー名 chikyu のユーザーのパスワードを入力します。) asano や sakura や dover でこれを行おうとすると “You are not in sudoers.” と怒られます。これは「あなたは apt でこのシステムのプログラムを変更する権限がありません」ということを意味しています。もしこれらのマシンにインストールしてほしいソフトウェアがある場合は admin210 までお問い合わせください。

次に、

```
$ apt search 欲しいソフトウェアのキーワード
```

としてパッケージ名を検索します。すでにパッケージ名を知っている場合は飛ばしても構いません。

最後に、

```
$ sudo apt install パッケージ名
```

としてインストールします。

他にも様々な機能がありますので、興味がある方は man ページを見るなりインターネット検索を駆使するなりしてみるといいでしょう。

### 7.2.2 Homebrew

macOS の場合は標準ではパッケージマネージャはインストールされていません。そのため自力でパッケージマネージャをインストールすることになるのですが、この際にほとんどすべての人が使っているものが Homebrew です。

https://brew.sh/index_ja を参考にインストールしましょう。

使い方は大まかには apt と似ています。ただし、パッケージのリストは常にオンラインから最新のものを取得するようになっていますので、都度 update する必要はなくなっています。

つまり、まず

```
$ brew search 欲しいソフトウェアのキーワード
```

としてパッケージ名を検索し、次に

```
$ brew install パッケージ名
```

としてインストールすればよいことになります。

## 7.3 UNIX コマンドチートシート

### 7.3.1 ディレクトリ・ファイル操作

| コマンド | 説明 |
|---|---|
| pwd | カレントディレクトリのパス (今いる場所) を表示する。 |
| cd [ディレクトリ] | ディレクトリへ移動する。<br>cd - 直前にいたディレクトリに戻る |
| ls [ディレクトリ] | ディレクトリの内容を一覧表示する。<br>主なオプション：-a「.」で始まる隠しファイルを含める、-l 詳しい説明、-F ファイルタイプ識別子 (ディレクトリは「/」、実行可能ファイルは「*」など) を表示、-R サブディレクトリ内もすべて表示、-h ファイルサイズをわかりやすい単位で表示、--color=always 色つきで表示 |
| mkdir [ディレクトリ] | ディレクトリを作成する。 |
| rmdir [空のディレクトリ] | 空のディレクトリを削除する。ディレクトリが空でない場合は削除できない。 |
| cp [コピー元] [コピー先] | コピー元のファイルをコピー先にコピーする。<br>-i 上書きするか確認する、-r コピー元ディレクトリを再帰的にすべてコピーする、-a アクセス権限・所有権やタイムスタンプなどを保持したまま再帰的にすべてコピーする<br>cp *file1 file2 … fileN targetdir* とすると、*file1 file2 … fileN* を *targetdir* にコピーできる。 |
| mv [移動元] [移動先] | 移動元のファイルを移動先に移動またはリネーム（名前の変更）する。ファイル名を変える場合は、移動先は同じディレクトリの別の名前にする。<br>-i 上書きするか確認する<br>mv *file1 file2 … fileN targetdir* とすると、*file1 file2 … fileN* を *targetdir* に移動できる。 |
| rm [ファイル] | ファイルを削除する。<br>-i 削除するか確認する、-f 存在しないファイルを無視し確認も行わない、<br>-r ディレクトリとその中身を再帰的に削除する<br>rm *file1 file2 … fileN* とすると、*file1 file2 … fileN* を削除できる。 |
| touch [ファイル] | ファイルのタイムスタンプを現在時刻に更新する。 |

| | |
|---|---|
| | ファイルが存在しない場合は空ファイルを作成する。 |
| ln [対象先] [リンク名] | (オプションなし) ハードリンク（ファイルの別名）を作成する。<br>-s シンボリックリンク（Windows のショートカット・Mac のエイリアス）を作成する |
| unlink [リンク名] | リンクを削除する。 |
| chmod [パーミッション] [ファイル・ディレクトリ] | ファイル・ディレクトリのアクセス権限 (パーミッション) を変更する。<br>モードビット：r(4) 読込み権限、w(2) 書込み権限、x(1) 実行権限、u(百) ユーザー（自分)、g(十) グループのメンバー、o(一) 他人、a 全員（デフォルト)、+ 付与、- 剥奪、= 設定<br>パーミッションの例：755、+x、g-w、go=rx、etc... |
| chown [ユーザー名] [ファイル・ディレクトリ] | ファイル・ディレクトリの所有権を変更する。 |
| find [評価式] | ファイル・ディレクトリを検索する。<br>評価式：-name [文字列] 名前が文字列にマッチするファイルを検索する（ワイルドカードを使用できる)、-size [サイズ] サイズより大きいファイルを検索する、などなど |
| tar | tar 形式アーカイブを作成・展開する。<br>tar cvzf [アーカイブ名] [ファイル・ディレクトリ] ファイル・ディレクトリを圧縮する（新しいアーカイブを作成し (c)、処理したファイルの一覧を出力し (v)、gzip 形式で圧縮し (z)、指定したアーカイブ・ファイルに出力する (f))[*4]<br>tar xvzf [アーカイブ名] 圧縮されたアーカイブを展開する（アーカイブからファイルを抽出する (x)) |

表 7.1: ディレクトリ・ファイル操作のコマンド

### 7.3.2 テキスト表示・操作

これらのコマンドはこれからよく使うので、覚えておいた方がよいでしょう。

| コマンド | 説明 |
|---|---|
| echo [文字列] | 文字列を標準出力（ターミナル）に出力する。-n 改行しない |
| cat [ファイル] | ファイルの内容を標準出力（ターミナル）に出力する。-n 行番号をつけて表示<br>cat *file1 file2 ... fileN* とすると、*file1 file2 ... fileN* を指定した順に連続して出力する。この機能を利用して、リダイレクト (>) と組み合わせれば 2 つ以上のファイルを連結できる。(むしろこっちがもとの使いかた。) |
| more [ファイル] | テキストファイルの内容を画面単位で表示する。<br>コマンド：[Space/z] 1 画面進む、[b] 1 画面戻る、[Enter] 1 行進む、[d] 半画面進む、[/] 検索、[n] 次候補、[v] エディタ起動、[q] 終了 |

[*4] tar はコマンドラインオプションに「-」(ハイフン) を入れなくてもよいという珍しいコマンドです。これは tar というコマンドが大昔からあるコマンドで、データを磁気テープに保存していた時代の名残です（f オプションを指定しないと、デフォルトで /dev/rmt0、すなわち磁気テープデバイスに出力されます)。その頃はオプションにハイフンを入れる習慣がなかったのです。現在ではハイフンを入れることもできますが (tar -cvzf)、ハイフンの有無で挙動が微妙に異なるので注意が必要です。例えば tar cfvz はうまくいきますが（オプションの順番が違ってもいい)、tar -cfvz はエラーになってしまいます。とりあえず、「tar cvzf」= ファイルを圧縮するコマンド、「tar xvzf」= ファイルを展開するコマンド、というように覚えておけばよいでしょう。(参考 URL:https://qiita.com/tatesuke/items/c5370823adc7772d55d8, https://qiita.com/junjis0203/items/6bb48184b508045e69da)

| | |
|---|---|
| less [ファイル] | テキストファイルの内容を画面単位で表示する。[*5] -N 行番号を行頭につけて表示<br>コマンド：[Space/f] 1 画面進む、[b] 1 画面戻る、[Enter/e] 1 行進む、[y] 1 行戻る、[d] 半画面進む、[u] 半画面戻る、[g] 先頭へ、[G] 末尾へ、[/] 検索、[n] 次候補、[N] 前候補、[v] エディタ起動、[q] 終了 |
| head [ファイル] | ファイルの先頭の部分を出力する。(オプションなしで 10 行) -n $n$ 指定した行数を出力 |
| tail [ファイル] | ファイルの末尾の部分を出力する。(オプションなしで 10 行)<br>-n $n$ 指定した行数を出力、-n +$n$ 指定した行数以降出力、-F 末尾監視 (ログファイルのチェックなどに使用) |
| grep [パターン] [ファイル] | ファイルのパターンにマッチする行を検索して出力する。(後述) |
| sed [スクリプト] [ファイル] | ファイルの文字列を編集して出力する。(後述) |
| awk [スクリプト] [ファイル] | ファイルの文字列の検知や処理を出力する。(後述) |
| diff<br>[ファイル 1] [ファイル 2] | **ファイル 1** と**ファイル 2** の内容の違いを比較する。<br>-c context 出力形式 -u unified 出力形式 -y side-by-side 出力形式 (2 画面) お好みでどうぞ |

表 7.2: 覚えておきたいテキスト操作のコマンド

これらのコマンドは覚える必要はありませんが、知っておくと便利です。

| コマンド | 説明 |
|---|---|
| tee [ファイル] | 標準入力（パイプなど）を標準出力（ターミナル）とファイルの両方に出力する。<br>-a ファイルに追記する |
| nkf [ファイル] | （オプションで与えた）文字コード、改行コードに変換する。<br>-j JIS -e EUC -s Shift-JIS -w,-w8 UTF-8 -w16 UTF-16 -x 半角カナ -X 全角カナ<br>-Lw CR+LF(Windows 改行コード) -Lm CR(Mac 改行コード) -Lu LF(UNIX 改行コード) |
| cut [ファイル] | ファイルの各行からオプションで指定した範囲を切り出す。<br>-c $n_1$–$n_2$ 文字数範囲 -f $n_1$–$n_2$ フィード数範囲 -d フィードの区切り文字 |
| rev [ファイル] | ファイルの各行の文字を逆に並べ替えて出力する。 |
| tac [ファイル] | ファイルの行を逆に並べ替えて出力する。cat の逆。 |
| sort [ファイル] | ファイルの行をソートして出力する。-r 逆順にする、-u 重複する行を 1 行にする |
| uniq [ファイル] | ファイルの重複する行を削除して出力する。予めソートしておく必要がある。<br>-c 重複の行数を表示する |
| wc [ファイル] | ファイルの行数・単語数・バイト数を表示する。<br>-l 行数を表示する、-w 単語数を表示する、-m 文字数を表示する、-c バイト数を表示する |
| seq [初期値] [間隔値] [最終値] | 等差数列を出力する。初期値・間隔値は省略可（デフォルトは 1） |
| tr | リダイレクトやパイプで入力された文字列から、指定された文字を削除・圧縮して出力する。<br>-d *'word' word* を削除する、-s *'ab' a* や *b* の繰り返しを 1 文字に圧縮する |

[*5] more は大昔からあるページャで、less は more を高機能にした拡張版として作られました。しかしその後 more 自体の機能が拡張され、現在では両者の機能の違いはわかりにくくなっています。挙動の違いとしては、more は末尾まで表示すると自動的に終了しターミナルに表示が残るのに対し、less は [q] を押すことで終了し表示が消えます。状況に応じて、好きな方を使えばよいでしょう。

| | |
|---|---|
| expr [式] | 式を計算・評価して出力する。(後述) |
| bc | 標準入力（パイプなど）で入力された計算式を数値計算して出力する。expr より高機能。 |
| nl [ファイル] | ファイルを行番号をつけて出力する。 |

表 7.3: 知っていると便利なテキスト操作のコマンド

### 7.3.3 プロセス・システム管理

ここには簡単な説明だけを書きます。覚えなくてよいです。実際に試して表示内容を確かめてみましょう。

| コマンド | 説明 |
|---|---|
| ps | プロセスの状態を表示する。プロセス ID、制御端末名、CPU 時間、実行ファイル名（コマンド）がわかる。ps aux などとするとより多くの情報が得られる。 |
| pstree | プロセスの親子関係をツリー形式で表示する。 |
| top | プロセスの状態をリアルタイムで表示する。[*6] [q] で終了。 |
| kill [プロセス ID] | プロセスを終了する。 |
| jobs | ジョブリストを表示する。 |
| bg [ジョブ ID] | 一時停止 (Ctrl+Z) したジョブをバックグラウンドで再開する。&を付け忘れたときに使う。 |
| sleep [時間] | 指定した時間だけ待機する。(デフォルトの単位は秒) |
| nohup [コマンド] & | ハングアップシグナルを無視してコマンドを実行する。サーバー上で時間がかかる計算や処理をするとき、ログアウトしたり ssh 接続が切れたりしても処理を続けることができる。 |
| date | 現在の日付と時刻を表示する。 |
| cal | カレンダーを表示する。 |
| which [コマンド] | コマンドのフルパスを表示する。 |
| whereis [コマンド] | コマンドのフルパスを全部表示する。 |
| alias [エイリアス]='[コマンド]' | コマンドに別名（エイリアス）をつける。~/.bashrc などに書いておく。引数なしでエイリアスを一覧表示する。 |
| unalias [エイリアス] | エイリアスを停止する。 |
| history | コマンド履歴を表示する。~/.bash_history などに保存されている。 |
| env | 環境変数の一覧を表示する。 |
| export [環境変数]='[文字列]' | 環境変数を設定する。 |
| script [ファイル] | ターミナルの入出力をファイル（デフォルトは./typescript）に保存する。 |
| w | ログインユーザーとプロセス名を表示する。 |
| who | ログインの状態を表示する。-b システムのブート時刻を表示する。 |
| hostname | ホスト名を表示する。<br>-f ホスト名 + ドメイン名を表示する。-i ホストの IP アドレスを表示する。 |

[*6] この画面を見てなんか面白そうだと思ったり、つい見とれてしまった人は、コンピュータ・オタクになる素質があるでしょう。もっとも、課題や研究などで大量のジョブを流した後や、多くの人が利用する計算機システム（スパコンなど）で top を眺めるととても楽しいです。

| | |
|---|---|
| free | メモリ使用量を表示する。-h わかりやすい単位で表示する |
| df | ディスク使用量とマウント位置を表示する。-h わかりやすい単位で表示する |
| du [ファイル・ディレクトリ] | ファイル・ディレクトリのサイズを表示する。<br>-h わかりやすい単位で表示する、-s ディレクトリの合計サイズを表示する |
| mount [デバイス][マウント位置] | ファイルシステムをマウント位置にマウントする。USB メモリなどをつなぐときに使う。<br>例：mount /dev/sdb /mount/usb デバイス名は dmesg などで確認する。 |
| umount [デバイス or マウント位置] | ファイルシステムをアンマウントする。USB メモリなどを取り外すときに使う。 |
| sl | キータイプ矯正。-l 長い、-a 助けを求める、-F 飛ぶ |

表 7.4: プロセス・システム管理のコマンド

### 7.3.4 ネットワーク操作

これらのコマンドはよく使うので、覚えておきましょう。

| コマンド | 説明 |
|---|---|
| ssh | SSH でログインする。（後述）ssh [コマンド] SSH でリモートホストでコマンドを実行する<br>-i 秘密鍵を指定する -p ポートを指定する |
| scp | SSH でファイルを転送する。（後述）-P ポートを指定する |
| wget [URL] | ファイルをダウンロードする。 |
| curl [URL] | ファイルをダウンロード・アップロードする。 |

表 7.5: ネットワーク操作のコマンド

**コラム:SSH/SCP 早わかり表**

- SSH ログイン (例:dover) 自宅から

```
~$ ssh s2126??@dover.eps.s.u-tokyo.ac.jp
```

地惑ネット (559 など) から

```
~$ ssh s2126??@dover
```

- SCP ファイル転送 (例:dover) ファイルを dover の (ホームディレクトリ) へ送る (自宅から)

```
~$ scp okurimono s2126??@dover.eps.s.u-tokyo.ac.jp:
```

ファイルを dover の (ホームディレクトリ) へ送る (地惑ネットから、以下同じ)

```
~$ scp okurimono s2126??@dover:
```

ファイルを dover の /mydir/へ送る

```
~$ scp okurimono s2126??@dover:mydir/
```

ファイルを dover の/home2/sensei/submit/へ送る

```
~$ scp okurimono s2126??@dover:/home2/sensei/submit/
```

カレントディレクトリ以外にあるファイルを dover の/home2/sensei/submit/へ送る

```
~$ scp /dir1/dir2/dir3/okurimono s2126??@dover:/home2/sensei/submit/
```

ファイルを dover の (ホームディレクトリ) からとってくる

```
~$ scp s2126??@dover:moraimono .
```

ファイルを dover の /mydir/からとってくる

```
~$ scp s2126??@dover:mydir/moraimono .
```

ファイルを dover の/home2/sensei/kadai/からとってくる

```
~$ scp s2126??@dover:/home2/sensei/kadai/moraimono .
```

ファイルを dover の/home2/sensei/kadai/からとってきてカレントディレクトリ以外に置く

```
~$ scp s2126??@dover:/home2/sensei/kadai/moraimono /dir1/dir2/dir3/
```

ディレクトリごと dover の (ホームディレクトリ) へ送る

```
~$ scp -r okurimono_dir s2126??@dover:
```

ディレクトリごと dover の (ホームディレクトリ) からとってくる

```
~$ scp -r s2126??@dover:moraimono_dir .
```

これらのコマンドは覚える必要はありませんが、知っておくと便利です。ここには簡単な説明だけを書きます。実際に試して表示内容を確かめてみましょう。

| コマンド | 説明 |
|---|---|
| ip | ネットワークインターフェイスを表示・設定する。ip addr（または ip a）IP アドレスを表示する。（ifconfig と同じ） |
| ss | ネットワークのソケットの状態を表示する。（netstat と同じ） |
| ping [IP アドレス/ホスト名] | パケットを送って応答を調べる（生存確認）。 |
| traceroute [IP アドレス/ホスト名] | ネットワークの経路を調べる。 |
| host [IP アドレス/ホスト名] | IP アドレスからホスト名を調べる／ホスト名から IP アドレスを調べる。 |
| nslookup [IP アドレス/ホスト名] | IP アドレスからホスト名を調べる／ホスト名から IP アドレスを調べる。 |
| whois [ドメイン名] | ドメインの登録情報を調べる。 |
| rsync [コピー元] [コピー先] | ファイルやディレクトリを同期する。ローカル同士でも使用可。 |
| mail | ターミナルでメールを送信する（メールサーバーでのみ機能します）。 |
| w3m [URL] | ターミナルで Web ページを閲覧する。 |

表 7.6: ネットワーク操作のコマンド

## 7.4 正規表現

正規表現とは、文字列のパターン（集合）を記述する方法です。文字とメタキャラクタを組み合わせることで、曖昧な文字列のパターンを表現することができます。前回出てきたコマンド grep、sed、awk も正規表現を用いて文字列を扱うことができます。ここでは代表的なメタキャラクタを紹介します。

ある文字列がある正規表現のパターンに該当することを「マッチする」と言います。words.txt というファイルには以下に出てくるような単語をリストアップしてあります。正規表現のパターンにマッチする文字列は、

```
~$ grep -E '[正規表現のパターン]' words.txt
```

マッチしない文字列は、

```
~$ grep -E -v '[正規表現のパターン]' words.txt
```

で出力することができます。以下の正規表現のパターンに対して、どの単語がマッチするか確かめてみましょう。もちろん、words.txt に何かしらの文字列を自分で追加しても構いません。

### 7.4.1 普通の文字

次節に挙げるメタキャラクタ以外の文字列は、その文字列を含む文字列にマッチします。

| 正規表現 | 一致する文字列の例 | 一致しない例 |
|---|---|---|
| ame | america, kame-san | AMeDAS, mae |

‘ame’ という文字列を持つ単語はマッチし、‘ame’ という文字列を持たない単語はマッチしません。

### 7.4.2 正規表現のメタキャラクタ

| メタキャラクタ | 説明 |
|---|---|
| . | 任意の 1 文字。 |
| ^ | （文字列の前に付けて）先頭が一致する。 |
| $ | （文字列の後ろに付けて）末尾が一致する。 |
| * | 直前の文字・パターンが 0 個以上続く。 |
| + | 直前の文字・パターンが 1 個以上続く。 |
| $\{n\}$ | 直前の文字・パターンが n 個続く。 |
| $\{n,\}$ | 直前の文字・パターンが n 個以上続く。 |
| $\{,m\}$ | 直前の文字・パターンが m 個以下続く。 |
| $\{n,m\}$ | 直前の文字・パターンが n 個以上 m 個以下続く。 |
| ? | 直前の文字・パターンが 0 個または 1 個である。 |
| [ ] | 角括弧内のいずれかの 1 文字である。 |
| [^ ] | 角括弧内のどの 1 文字も含まない。 |
| - | 角括弧内で、連続する文字を指す。（例：a-z,0-9） |
| \| | いずれかに一致する。 |

表 7.7: 正規表現のメタキャラクタ

なお、メタキャラクタに使用されている記号を、メタキャラクタとしてではなく記号自体として用いたい場合はエスケープシーケンス\ を用いて*などとします。\ 自身を表したい場合は\\ とします。

ここで注意してほしいことは、**正規表現のメタキャラクタはシェルのメタキャラクタとは全く別物である**、ということです。[*7]例えば * は、シェルでは「任意の文字列」という意味でしたが、正規表現では「直前の文字・パターンが 0 個以上続く」という意味です。混乱しないように気をつけてください。

以下にメタキャラクタの使用例を示します。

[*7] このことはときに厄介な問題を引き起こします。すなわち、メタキャラクタによっては、人間が正規表現のメタキャラクタのつもりで入力したものが、引数としてコマンド（grep など）に渡される前に、シェルによってシェルのメタキャラクタとして解釈されてしまい、結果として人間の意図しない動作になり、エラーになって返ってくることがあります。例えば | （バーティカルライン）は、grep に「いずれかに一致する、という意味の正規表現」として渡される前に、シェルによって「パイプ」として解釈されてしまします。この問題を防ぐために、常に正規表現を含む引数全体を '' （シングルクオーテーション）で囲むとよいでしょう。

| 正規表現 | 一致する文字列の例 | 一致しない例 | 説明 |
|---|---|---|---|
| ^ame | america | name | 語頭が ame かどうか |
| ame$ | name | america | 語尾が ame かどうか |
| ame.i | america, samejima | chameleon | 「ame{任意の 1 文字}i」という文字列があるかどうか |
| am*e | name, amme | 551umeeee | 「a{m が 0 個以上}e」という文字列があるかどうか |
| a[mn]e | america, plane | karaage, ase | 「ame」または「ane」という文字列があるかどうか |

表 7.8　メタキャラクタの使用例

### 7.4.3　メタキャラクタの組み合わせ

普通の文字とメタキャラクタをうまく組み合わせて使いましょう。

| 正規表現 | 一致する文字列の例 | 一致しない例 | 説明 |
|---|---|---|---|
| a[m-z]e | name, plane | karaage, anone | 「a{m〜z のうちの 1 文字}e」<br>という文字列があるかどうか |
| am.*e | name, ambulance | plane | 「am{0 文字以上}e」<br>という文字列があるかどうか |
| ame[m-z]* | america, sasameyuki | karaage | 「ame{m〜z のみを使った 0 文字以上}」<br>という文字列があるかどうか |
| ame[m-z]*$ | name | chameleon | 「ame{以降 m〜z のみを使った 0 文字以上}」<br>で終わる文字列かどうか |
| ame[m-z]+ | america | amedas | 「ame{m〜z のみを使った 1 文字以上}」<br>という文字列があるかどうか |
| ame[m-z0-5]+ | america, ame334 | ame, ame765 | 「ame{m〜z,0〜5 のみを使った 1 文字以上}」<br>という文字列があるかどうか |

表 7.9: 組み合わせの利用例

### 7.4.4 grep, sed, awk で正規表現を使う

前回の授業で grep, sed, awk について勉強しました。復習しながら、これらのコマンドで正規表現を使う方法を見ていきましょう。

grep

| コマンド | 説明 |
|---|---|
| grep [パターン] [ファイル] | ファイルのパターンにマッチする行を検索して出力する。<br>-v パターンに一致しない行を表示する、-c パターンに一致する行数を表示する、-n 行番号を表示する、-m $n$ n 回マッチしたら終了する、--color=always 一致したパターンに色を付ける |

ファイルや標準入力を読み込み、パターンにマッチする行を検索して出力します。正規表現でパターンを指定するときは、grep -E ' 正規表現' ファイル とします。[*8]

sed

| コマンド | 説明 |
|---|---|
| sed [スクリプト] [ファイル] | ファイルの文字列を編集して出力する。<br>-n コマンド p で指定された行以外出力しない、-e スクリプトを追加する、-i ファイルを上書きする |

sed は”Stream EDitor” の略でエディタの一種です。ファイルや標準入力を読み込み、テキストを編集し、標準出力に出力します。オプション-i を指定しない限り、ファイルが編集されることはありません。

sed は、「スクリプト」で処理を指定します。「スクリプト」は「アドレス」と「コマンド」からなります。

まず「アドレス」から見ていきます。「アドレス」は処理する行を指定します。

| アドレス | 説明 |
|---|---|
| $n$ | n 番目の行。 |
| $ | 最終行。 |
| /正規表現/ | 正規表現にマッチする行。 |
| (アドレス 1),(アドレス 2) | アドレス 1 からアドレス 2 まで。 |
| (アドレス 1),+$m$ | アドレス 1 から m 行先まで。 |
| (アドレス 1)~$m$ | アドレス 1 から m 行毎に。 |

表 7.12: sed のアドレス

[*8] -E は「パターンに拡張正規表現を使う」というオプションです。正規表現は長い歴史の中で、「基本正規表現」→「拡張正規表現」→「Perl の正規表現」というように増改築されていきました。実は先ほど勉強したのは「拡張正規表現」です。しかし grep や sed のデフォルトは「基本正規表現」で、残念なことに「拡張正規表現」と互換性がありません。したがって「拡張正規表現」を使用するには、grep ではオプション-E またはコマンド egrep、sed ではオプション-r を使用する必要があります。awk はデフォルトで「拡張正規表現」が使えるのでオプションは必要ありません。

次に「コマンド」を見ていきます。アドレスで指定した各行に対する処理内容を指定します。

| コマンド | 説明 |
|---|---|
| = | 行番号を表示する。 |
| a [テキスト] | 行の下にテキストを追加する。 |
| i [テキスト] | 行の上にテキストを挿入する。 |
| c [テキスト] | 行全部をテキストで置換する。 |
| d | 行を削除する。 |
| p | 行の処理結果を出力する。 |
| s/パターン/文字列/ | 行中に” パターン” にマッチする文字列があれば” 文字列” に置換する。<br>行の中のマッチ最初の 1 つ目だけを置換する。 |
| s/パターン/文字列/g | 行中に” パターン” にマッチする文字列があれば” 文字列” に置換する。<br>行の中のマッチするすべてを置換する。 |

表 7.13: sed のコマンド

最後に「アドレス」と「コマンド」の組み合わせの具体例を見てみます。「アドレス」は省略すれば全行に対して処理されます。

| 具体例 | 説明 |
|---|---|
| `sed '1d'` | 1 行目を削除する。 |
| `sed -n '7,10p'` | 7 行目から 10 行目だけを表示する。 |
| `sed '/ame/a rain'` | ”ame” を含む行の下に”rain” を追記する。 |
| `sed 's/ame/kaze/'` | ”ame” を”kaze” に置換する。 |
| `sed -r '2,$s/a[ms]e/kaze/'` | 2 行目以降の”ame” と”ase” を”kaze” に置換する。 |
| `sed -r '/a/s/(ame\|yuki)/kaze/g'` | ”a” を含む行の”ame” または”yuki” をすべて”kaze” に置換する。 |
| `sed -r 's/^/- /'` | 行頭に”- ” を追加する。 |
| `sed -r 's/$/ -/'` | 行末に” -” を追加する。 |

表 7.14: sed の具体例

awk

| コマンド | 説明 |
|---|---|
| awk [スクリプト] [ファイル] | ファイルの文字列の検知や処理を出力する。<br>-F 区切り文字を指定する（デフォルトは空白） |

awk はファイルや標準入力を読み込み、空白などで区切られたデータを行単位で処理・整形し、標準出力に出力します。awk のスクリプトの基本構文は' 条件 {処理}' です。条件には”<” や”>” などを用いた関係式のほか、”/パターン/” として正規表現を使うことができます。処理には”print” などの awk の内部コマンドを利用します。

awk には組み込み変数が用意されています。”NR” は行数、”FILENAME” はファイル名、”NF” はフィールド（列）数を表します。また行全体は”$0” で、各フィールド（列）は”$1”,”$2”,··· という変数で表されます。

awk はプログラミング言語として拡張されていき、今ではさまざま処理ができるようになっています。

| 具体例 | 説明 |
|---|---|
| `awk '{print $5}'` | 5 列目を表示する。 |
| `awk 'NR<=10 {print}'` | 10 行目までを表示する。（`print $0` としても同じ） |
| `awk '$1>=100 {print $2,$3,$4}'` | 1 列目が 100 以上ならば、2,3,4 列目を表示する。（カンマを忘れない） |
| `awk '$2~/ame/ {print $2,$3,$4}'` | 2 列目が”ame” を含むならば、2,3,4 列目を表示する。 |
| `awk '$1>=100 && $2~/a[ms]e/ {print $2,$3,$4}'` | 1 列目が 100 以上かつ、2 列目が”ame” または”ase” を含むならば、2,3,4 列目を表示する。（or は||、not は!） |

表 7.16: awk の具体例

**練習問題**

stations.txt（丸ノ内線の駅一覧）から、次を表示するコマンドを考えてください。

(1) ”sanchome” が駅名に含まれる駅の数。
(2) Chiyoda Line と乗り換えができる駅のリスト。
(3) 駅間が 1.0km 未満の駅の組み合わせリスト。

そのほか自分で問題を作って、解いてください。

## 7.5 設定ファイルをいじってみよう

中身が少し分かったところで、少しずついじってみましょう。ただし、特にシェルや X Window System 関連の設定ファイルは、下手にいじってしまうとログインできなくなる、という事態に陥りかねませんので、変更する場合は十分に気を付けましょう。また.emacs も Emacs 以外のエディタが使えない状況でおかしなことになってしまうと、修正することすらできなくなりますので、変更はある程度の知識や技術を身につけた後に行うのが無難かと思います。あとは自由にカスタマイズをして、便利に端末を使いましょう。

### 7.5.1 .ssh

これまでに触れた以外にも SSH のオプションの数だけ設定ファイルに書き込める項目があります。たとえば、XForwarding などがあります。また、先の~/.ssh/config の例で説明のなく使っていた ProxyCommand も非常に便利です。「ssh 踏み台」等のキーワードで検索するよいでしょう。どのような項目が設定できるか調べてみて、便利に使えるようにカスタマイズしていきましょう。

### 7.5.2 .bashrc

自分で環境を構築していく場合以外は、このファイルを大きくいじることは無いと思います。そもそも、管理者が端末の環境に応じて設定している部分が多いので、中を見てみて、気になる部分を調べてみましょう。皆さんのホームディレクトリにある.bashrc は、コメントも付いていますので、参考にしてみてください。ちなみに、シェル起動中に.bashrc のような設定ファイルを読み込むには、ターミナル上でsource .bashrcなどとします。先ほどから何度か出てきましたね。

### 7.5.3 .aliases

エイリアスは、どんどん自分用にカスタマイズして、便利に使いましょう。書式は、

```
alias  ll='ls  -l'
```

といった具合です。“=” の前後に空白を入れないようにしてください。これは別に.bashrc や.aliases に書き込まなくてはいけないわけではなく、ターミナル上で設定できます。

```
~$ alias ls='ls -R'
```

```
~$ alias ls='sl'
```

とすれば、現在のシェル (ターミナル) が動いている間はこの設定が有効となります。しかし、これでは現在のシェルを終了させてしまうと設定が無効になり、次に起動させたシェルで再び入力をしなくてはなりません。.bashrc 内の設定も同様ですが、設定自体はターミナル上でコマンドを用いて行えるのですが、設定ファイルが読み込まれることで、このような手間を省いてくれているわけです。ちなみに、下側のやつは多分しない方がよい (ls が sl になってしまいます!!) です。alias をやり直したり、.bashrc を再読み込みするなどして、元に戻しておきましょう。

### 7.5.4 .emacs

Emacs は非常に拡張性に優れたエディタです[*9]。この Emacs の設定を行っているのが.emacs です。先ほど、中身をのぞきましたので、無難な部分として、初期メッセージの表示・非表示やメニューバーなどの表示、背景色などを変更してみましょう。初期メッセージに関しては ‘;’ でコメントアウトしてやればよいと思います。エディタで書き換えたら、Emacs を再起動してやってください。

ちなみに、.emacs　の中に

```
(set-foreground-color "white")
(set-background-color "dark green")
(set-cursor-color "yellow")
(set-mouse-color "white")
```

と書き足すと、配色が黒板のようになります。

TEX やプログラミングをやり始めると、こういうのも便利かもしれません。

[*9] メーラとしての機能やシェルモードなどの存在を考えると、ただのエディタとも言えませんが。弱点は若干重いことでしょうか。

```
;; 対の括弧を明示する(for remark of the other paren)
(show-paren-mode t)
```

その他、いろいろなカスタマイズ例がネット上には転がっていますので、調べて、使ってみましょう。