コンピュータリテラシ＃7 – フィルタとシェルスクリプト

久野 靖 (電気通信大学)

2017.5.19

1　今回の目標

今回の目標は次の通りです。

- フィルタを用いた柔軟なユティリティの構成について理解する— Unix で多様な作業をこなす原動力がユティリティです。
- シェルの変数や展開機能、実行パスについて学ぶ — Unix を使う上ではこれらの概念を知っていると効率が高められます。
- シェルスクリプトの考え方について理解する — 繰り返しおこなう作業をスクリプトにすることで作業効率がいっそう高くなります。

2　ユティリティとフィルタ

2.1　「大きなユティリティ」と「小さなユティリティ」 exam

ユティリティとはコンピュータの世界では「汎用的な作業をこなすプログラム」を意味します。そしてユティリティには「大きなユティリティ」と「小さなユティリティ」があります。大きなユティリティとは、アーミーナイフ (図 1 右) のように、1 つのプログラムが多数の機能をこなすようなものをいいます。便利そうですが、次の弱点があります。

- 指定が沢山必要で、使い方が複雑になりがちである。
- 1 つの機能だけ使いたくても大きなプログラムが動くので遅くなりやすい。
- 機能を増やしたり修正するのが大変である。

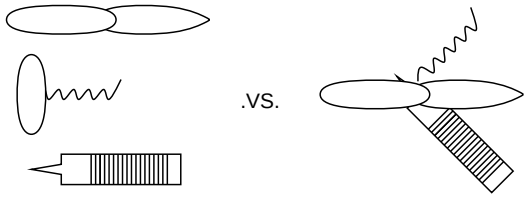

図 1: 単機能の道具と多機能の道具

一方小さなユティリティとは、個別の道具 (図 1 左) のようなもので、1 つのプログラムは単純な機能だけを提供し、複雑なことはプログラムを組み合わせて対応する考え方です。この方式では、大きなユティリティの弱点が克服できます。

- 1 つのユティリティはごく単純で使い方もすぐわかる。
- 使いたい機能に対応するユティリティだけ動かせば済む。
- 足りない機能があったら、その機能だけを行う小さなプログラムを書いて追加すれば既存のユティリティと組み合わせて使える。

2.2 フィルタ exam

Unix は伝統的に小さいユティリティの文化です。小さいユティリティのためには複数のプログラムを組み合わせて動かす仕組みが必要ですが、それがパイプラインです。

プログラム | プログラム | … | プログラム

パイプラインの途中にあるプログラムはどれも「標準入力から入力データを読み取り、処理を行なって、標準出力にデータを書き出す」形で動作します。その形がちょうど、空気や水をろ過するフィルタに類似しているので、この種のプログラムを Unix では**フィルタ** (filter) と読びます。既に学んだ cat もフィルタの 1 つです。

このやり方がうまくいくためには、どのプログラムの出力も別のプログラムの入力として使える必要があります。そのためには色々な考え方がありますが、Unix の場合は「テキストファイル」つまり人間が読み書きできるようなデータを扱う、という形で共通化しています。以下では代表的なフィルタをいくつか見ながら、これらの原理がどのように具体化されているかを学んでいきます。

2.3 tr — 文字の置換 exam

tr は入力の各文字を (原則として)1 対 1 で別の文字に変換 (TRanslate) して出力するフィルタです。キーボードが壊れていて小文字の「a」が大文字になってしまうマシンでファイルを打ち込んだとしましょう (何てわざとらしい例!)。あとで大文字の「A」を小文字に直すのには次のようにします。

```
...$ cat test.txt
ThAt is A cAt.
...$ tr A a <test.txt
That is a cat.
```

tr の基本的な使い方は次のとおりです。

- tr 文字列$_1$ 文字列$_2$ — 文字を別の文字に変換

「文字列$_1$」の 1 文字目は「文字列$_2$」の 1 文字目、2 文字目は 2 文字目、というふうに対応する文字どうしの変換が行われるわけです。ほかの多くのフィルタと違って、tr はファイル名指定を受け付けないので、ファイル入力が必要なら入力リダイレクションを用います。

しかし、注意深い人は「そのキーボードは小文字の「a」が入らないはず」と気づいたかも知れません。別マシンに移ったことにもできますが、壊れたマシンでやりたければ、任意の文字を 8 進文字コードで指定することができます (文字と 8 進コードの対応は「man ascii」で見てください)。

```
...$ tr A '\141' <test.txt
That is a cat.
```

もうちょっと実用的なのは、すべての大文字を対応する小文字に直すことでしょう。

```
...$ tr A-Z a-z <test.txt
that is a cat.
...$ tr ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz <test.txt
that is a cat.
```

どちらも同じ動作ですが、「-」で文字範囲を指定する方が楽ですね。ここまでは指定する 2 つの文字列の長さが同じでしたが、もし文字列 2 が短ければ、文字列 1 のあぶれた文字には全て文字列 2 の最後の文字が対応します。

```
...$ tr A-Za-z a <test.txt
aaaa aa a aaa.
```

tr にもいくつかのオプションがあります。

- -c — 文字列 1 に「ない」すべての文字を集めたものを改めて文字列 1 だと思う。

例を見てみましょう。

```
...$ tr -c A-Za-z / <test.txt  ←英字以外をすべて「/」に
ThAt/is/A/cAt//...$ ←改行が「/」になったので改行されない
```

「すべての」文字に改行文字も含まれるのが不都合なら、次のように改行も除外すればよいです。

```
...$ tr -c 'A-Za-z\012' /<test.txt  ← 012 は改行の 8 進コード
ThAt/is/A/cAt/
...$           ←改行されてから次のプロンプト
```

- -d — 文字列 1 に現われる各文字を消去 (delete)。この場合、文字列 2 は指定する必要がない。

こちらは不要な文字を削除するのに便利です:

```
...$ tr -d ' ' <test.txt  ←空白文字をすべて削除
ThAtisAcAt.
```

- -s — 置き換えが連続して起こる場合には、最初の 1 個だけ出力 (squeeze)。

たとえば単語の「数だけ」知りたい場合は、次のようにできます。

```
...$ tr -s A-Za-z a <test.txt
a a a a.
```

これらの指定を組み合わせて様々なことができます。たとえば「tr -cd 'A-Za-z\012'」で英字と空白と改行以外をすべて消して「きれいな」単語だけにできますし、「tr -cs A-Za-z '\012'」で英字以外の並びをすべて改行 1 個に置き換えて各単語を 1 行ずつバラバラにできます。

2.4 sort と uniq — 整列と重複除去 exam

ここから先で説明するフィルタはどれも、ファイルを指定すればそのファイルから読み込み、指定しなければ標準入力から読み込みます。**sort** は、ファイルの行を指定した順番に並べ替えてくれます。

- sort ファイル… — 行単位での並べ替え

いちばん簡単には、何もオプションを指定しないと、sort は行全体を比較して、その文字コード大小順にもとづき、行を小さい順に並べます:

```
...$ cat test.txt
this
is
a
pen
...$ sort test.txt
a
is
pen
this
```

しかし、文字コード順だと $A < B < \ldots < Z < a < b < \ldots < z$ なので、大文字と小文字が混ざっているとあまり嬉しくないかも知れません。

```
...$ cat test.txt
This
is
a
pen
...$ sort test.txt
This
a
is
pen
```

このようなときには「-f」(大文字小文字統合) オプションを指定すれば、対応する大文字と小文字の文字は同一とみなしてくれます。また、数値データも文字コード順で比較されるとあまり嬉しくありません。

```
...$ cat test.txt
10
2
1
...$ sort test.txt
1
10
2
```

文字コード比較だと「1」で始まるものが全部終わってからはじめて「2」で始まるものが来ます。このような場合には「-n」(number) オプションを指定すると、数値としての順に並べられます。いずれの場合も、「-r」(reverse:逆) オプションを追加すると小さい順でなく大きい順に並べられます。

行全体ではなく、行の特定の部分にもとづいて並べ変えを行わせることもできます。その指定方法は少し面倒ですが、いちばん簡単には「-k 1」「-k 2」などと指定することで (空白で区切られた)「1 番目の欄」「2 番目の欄」などを指定できます (詳しくは「man sort」を見てください)。

ふたたび、行全体を整列する場合に戻りますが、「どのような行があるか」だけ知りたい場合には重複を除く (同じ行が複数あった場合に 1 行だけ残してあとは消す) ほうが望ましいですね。一般のファイルについてこれをやるのは面倒ですが、整列後なら同じ内容の行が隣り合っているので簡単です。それをやってくれるのが **uniq** というフィルタです。

- uniq ファイル… — 整列後のファイルの重複を除く

たとえば単語リストでこれを行ってみましょう。

```
...$ cat test.txt
this is a pen.
what is this?
...$ tr -cs A-Za-z '\012' <test.txt
this
is
a
pen
```

```
what
is
this

...$ tr -cs A-Za-z '\012' <test.txt | sort
a
is
is
pen
this
this
what
...$ tr -cs A-Za-z '\012' <test.txt | sort | uniq
a
is
pen
this
what
```

なお、uniq に-c(count) オプションを指定すると、同じ行がいくつずつあったか数えてくれます:

```
...$ tr -cs A-Za-z '\012' <test.txt | sort | uniq -c
 1 a
 2 is
 1 pen
 2 this
 1 what
```

2.5 head と tail と wc — 先頭/末尾/数える exam

あと少しだけ、行数関係のフィルタについて簡単に説明します。

- head -行数 — 先頭から指定した行数のみ取り出す
- tail -行数 — 末尾から指定した行数のみ取り出す

これらはファイルの先頭 N 行または末尾 N 行だけを取り出すフィルタで、長いファイルの頭だけ/終わりだけ見たい場合に使います。行数を省略すると 10 行分取り出されます。また、入力の行数が指定行数より少ない場合はその全部が取り出されます。

また、入力の文字数や行数などが知りたい場合には、**wc** (Word Count) を使うことができます。

```
...$ ps | wc
    7     39       241
行数↑     ↑単語数   ↑文字数
```

数字が 3 つ表示されますが、順に「行数」「単語数」「文字数」です。「wc -l」「wc -w」「wc -c」の各オプションにより、行数、単語数、文字数だけを出力させることもできます。

演習 1 まず「echo This is A cAt — tr A z」など echo とパイプを使って tr の説明されている機能をひととおり確認。次は英語の文書が欲しいで、「setenv LANG C」を実行する (man を実行してみて英語になったことを確認。logout すると元に戻る)。「man コマンド名 >ファイル」でマニュアル内容をファイルに保存できるのでやってみる。もしファイルに行番号がついている方がよければ「cat -n ファイル」でできる。以上をふまえて、以下の課題をやってみなさい。

a. 行数の多いファイルの途中 (たとえば 100 行目から 110 行目まで) を取り出して表示するにはどうしたらいいか考えてやってみる。
b. 自分が選んだコマンドの man に出て来る単語の一覧 (出現回数つき) を作ってみる。たとえば「the」や「that」などのよく出て来そうな単語の出現回数は何回か調べてみる。
c. 自分が選んだコマンドの man に出て来る単語の出現頻度トップ 10 の表 (のようなファイル) を作成してみる。

3 正規表現

3.1 grep 族 — パターンにあてはまる行を探す exam

grep、**fgrep**、**egrep** の 3 つのコマンドはいずれも「入力のなかに指定したパターン (Unix の用語では**正規表現** (regular expression) にあてはまる行があったら、その行全体を打ち出す」という機能を提供する同類のフィルタです (指定方法は 3 つとも同じ)。

- grep パターン ファイル… — ファイル中でパターンを含む行を出力

オプションも次のものが 3 つ共通に使えます。

- -v — パターンを「含む行」の代わりに「含まない行」を打ち出す。
- -n — 行を打ち出す際に行番号を一緒に打ち出す。

そして、3 つの違いはパターンとして何が書けるかの違いになります。まず fgrep の場合、パターンとしてたんなる文字列のみが書けます。たとえば「that」をいつも「taht」打ってしまうくせがある人が、そのまちがいをチェックしたければ次のようにします:

```
...$ fgrep taht wrong.txt
What is taht?
```

しかし、「That」のように文の頭にくるときは大文字なのでこれではみつかりません。そのような場合には grep のパターンを使えばよいのです:

```
...$ grep '[Tt]aht' wrong.txt
Taht is a cat.
What is taht?
```

「[…]」は「…」の部分のどれか 1 文字にマッチするパターンです。なお、「[」と「]」はシェルのメタキャラクタ (後述) なので、grep にパターンとして渡すときには「'…'」で囲む必要があります。では、fgrep の存在意義は何でしょうか? それは、たとえば「[」という字を探したければ fgrep で探すほうが簡単なわけです (grep で探したい場合には「\[」のように前に「\」をつけなければなりません)。

さて、that だけでなく this も thsi と打ってしまう人が両方探したい場合はどうでしょうか。そのときは grep でも力不足で、egrep で次のように指定します:

```
...$ egrep '[Tt](aht|hsi)' wrong.txt
Taht is a cat.
What is taht?
Thsi isn't a dog.
```

表 1: grep 族のパターンのまとめ

パターン	説明
c	c という文字そのもの。
[⋯]	⋯のうちどれか (1 文字 (tr 同様 a-z のようにも書ける)。
.	任意の 1 文字。
^α	行の先頭の α
α $	行の末尾の α
α ?	α または空。(1)
α *	α というパターンの 0 個以上の繰り返し。(1)
α +	α というパターンの 1 個以上の繰り返し。(1)
(⋯)	くくり出し。(2)
α | β	α または β 。(2)
\(⋯\)	⋯のところを一時的に覚える。(3)
\1、\2	覚えたものの 1 番目、2 番目、⋯。(3)

丸かっこは「くくり出し」を、縦棒は「または」を表わしています。この丸かっこと縦棒が egrep で加わった機能なわけです。表 1 にパターンについてまとめておきます。

(1) は grep ではパターン α が 1 文字に対応するパターンでなければならないという制約があります。(2) は egrep のみの機能です。一方 (3) は grep のみの機能です。また、「.」の長い連続など、組み合わせが爆発的に多くなるパターンは egrep では実現上うまく扱えません。というわけで、grep と egrep は適材適所で使い分ける必要があるわけです。

興味深い練習として、/usr/share/dict/words という英単語が多数入ったファイルからパターンに合った単語を取り出して見ましょう (less を使うのは、多数見つかったときゆっくり見るため)。

```
...$ grep 'パターン' /usr/share/dict/words | less
```

パターンの例をあげておきます (うろ覚えの単語をさがすという実用的な使い方も可です)。

'[aeiou][aeiou][aeiou]'	母音が 3 つ続く単語
'tion$'	末尾が tion で終わる単語
'^z'	z で始まる単語
'^..........$'	長さ 10 文字の単語
'\(...\)\1'	3 文字の反復を含む単語
'\(.\)\(.\).\2\1'	5 文字の回文を含む単語

3.2 sed — 文字列を置き換える exam

tr による文字置換は強力ですが、「自分は that を taht と打ってしまうので、これを正しく直したい」のような仕事には無力です。tr は各文字をバラバラに扱うので「特定の文字の並び」には対応できないからです。そのような文字列の置き換えには **sed**(stream editor) が適役です。

- sed コマンド ファイル⋯ — コマンドに従って入力を加工する

たとえば上の例題は次のようにしてできます:

```
...$ cat wrong.txt
What is taht?
...$ sed 's/taht/that/' wrong.txt
What is that?
...$
```

この「s」(substitute) コマンドだけ覚えておけばほとんど十分でしょう (他のコマンドは man sed で見てください)。なお、上の s コマンドは 1 行に 1 回しか置き換えを行ないませんが、taht が全部 that になるまで繰り返したければ「's/taht/that/g'」のように末尾に「g」をつけてください。

たったこれだけ…? と思うかもしれませんが、「s」コマンドによる指定には grep と同じパターンが書けるので、これだけでかなり強力な修正ができます。例を見てみましょう。

```
...$ cat test.txt
a 21
is 10
this 3
...$ sed 's/\(.*\) (.*\)/\2 \1/' test.txt
21 a
10 is
3 this
```

これは、「入力行を任意の文字列 1 と、空白と、また別の任意の文字列 2 にマッチさせ、それ全体を 2、空白、1 の順でつなげたものに置き換える」わけです。

場合によっては、こういう置き換えを多数やりたいかもしれません。その場合は

```
sed -e 's/Thsi/This/g' -e 's/Taht/That/g'
```

のように各コマンドのまえに-e オプションをつければ、いくつでもコマンドが書けます。あるいは、

```
s/Thsi/This/g
s/Taht/That/g
```

のように多数のコマンドを並べたファイルを準備しておき、「sed -f ファイル」の形で指定することもできます。sed は入力の各行についてファイルのなかにある命令を 1 行ずつ「実行」してくれます。つまり、これは一種の「プログラム」なわけです。

演習 2 正規表現を扱うフィルタの課題をいくつか挙げておきます。

a. /usr/share/dict/words から次のような単語を探しなさい (3 つ以上やってみること)。

- 「aho」というつづりと「ya」というつづりが両方含まれている単語。[1]
- 末尾が「otion」で終わる単語で、「e」が含まれないようなもの。[2]
- 先頭が「z」で最後が「tion」で終わる単語。[3]
- 母音を 5 つ連続して含む単語。[4]
- 5 文字のまったく同じ文字の並びが 2 回出て来る単語。[5]

b. sed を使って以下のことをやりなさい (2 つ以上やること)。

- This や this を Thsi や thsi、That や that を Taht や taht と打ってしまう可哀想な人の間違いを修正する。[6]
- This や this をすべて That や that に、逆に That や that をすべて This や this に一括して修正する。[7]
- ファイルの各行の頭にある空白の数を 2 倍にする (0 個なら 0 個、1 個なら 2 個、2 個なら 4 個…)。[8]

[1]ヒント: まず「aho」が含まれているものを取り出し、その出力の中からさらに「ya」が含まれているものを探します。
[2]ヒント: 「終わる」は、grep の$の機能を使います。その後でパイプで「-v」つきの grep で「e」を排除すればよい。
[3]ヒント: 途中に任意文字が 0 個以上あるわけです。
[4]ヒント: 母音とは「a」「e」「i」「o」「u」のどれかですね。
[5]ヒント: 5 文字の並びを覚えて、そのあとに任意文字が 0 個以上あり、その後で覚えた並びがあればいいですね。
[6]ヒント: sed の出力をパイプでまた sed に接続することで、いくつもの置き換えを指定できます。
[7]ヒント: 最初の置き換え先を「%%%」等の目印にしておき、2 番目の置き換え後に本来のものに置き換え直します。
[8]ヒント: 行頭にある空白の列を覚えて 2 回出力すればできます。

- `ls -l` の出力からファイル名前と大きさ (バイト数) の部分だけ抜き出し表示する。[9]
- 上と同じだが、ただし名前が左側、バイト数が右側に来るようにして、なおかつバイト数を右そろえする。[10]

c. (チャンレンジ問題)「`aaaaabbbcc`」のように、「a が N 個、b が M 個、c が T 個」並んだ行について「できるだけ多くの a の並びと b の並び (a と b 同数つまり $min(M, N)$ 個) を消し、その個数だけ c の並びに追加する」置き換えを `sed` で作りなさい。[11]

4 シェルの進んだ機能とシェルスクリプト

4.1 シェル変数と変数展開

変数 (variable) とは (コンピュータ業界では) おもにプログラミングの用語であり、値を入れておける (そして変化させられる)「入れ物」「箱」のようなものを意味します。そして、シェルにもそのような意味での変数があります。このあたりはシェルの種別によって違うのですが、ここでは sol で標準に設定されている `tcsh` を前提とします。

シェルでは変数は値を入れたときに作られます。変数の名前は英字 (A〜Z, a〜z, _) で始まり、英数字が並んだものである必要があります。変数に値を入れるのは `set` コマンドを使います。

```
...$ set ax1 = 'abc'
```

次に変数から値を取り出すときは、**変数展開** (variable substitution) を使います。具体的には何かというと、「`$`変数名」という形のものがあると、その部分が変数の中身で置き換わるのです。これを確認するのに便利なコマンドが `echo` です。

```
...$ echo "$ax1 + $ax1"
abc + abc
```

文字列の中の「`$ax1`」が先程入れた「`abc`」に置き換わっています。なお、この例のように文字列`"..."`の中では変数展開は置きますが、`'...'` で囲んだ場合は置きません。また、何も囲まない場合はもちろん展開が起きます。1 つ注意するのは、変数のうしろにすぐ英数字がくっついていると変数名が変わってしまうので、そのような場合は変数名を「`{...}`」で囲む必要があるということです。

```
...$ echo '$ax1 + $ax1'
'$ax1 + $ax1'          ← '...' の中は変数展開なし
...$ echo This is $ax1
This is abc            ← 文字列の外は当然変数展開
...$ set ax1x = 'def'  ← 新たな変数に値をセット
...$ echo ${ax1}x is not $ax1x
abcx is not def        ← 2 つの変数は別
```

あと、変数に入れる値はリスト (かっこで囲んだ値の並び) であってもよいです。このときは全体として取り出す以外に「`$`変数名 `[`番号`]`」で特定の要素だけを参照もできます。

```
...$ set a = (x1 x2 x3 x4) ←リストを入れる
...$ echo $a
x1 x2 x3 x4          ←表示では () は現れない
...$ echo $a[2]      ←2 番目だけ参照
```

[9] ヒント: 地道に名前や大きさの部分だけ取り出すパターンを作るだけです。
[10] ヒント: 揃えるには、十分多くの空白にしてから、長すぎるものを削除します。
[11] ヒント: a と b の個数に上限を決めることにすれば、その数だけ`-e` を使ってコマンドを並べる (か、またはその数だけパイプラインでつなげる) ことでできます。上限を決めない方法については「`mand sed`」を熟読する必要あり。

```
x2
...$ set a[3] = zzz  ←3番目を変更
...$ echo $a
x1 x2 zzz x4         ←確かに変化した
```

4.2 既定義なシェル変数と実行パス

シェル変数の中には最初から値が入っているものがあります。代表的なものを挙げておきます。

- `$user` — 自分のユーザ名が入っている
- `$home` — 自分のホームディレクトリの絶対パス名が入っている
- `$cwd` — 現在位置の絶対パス名が入っている
- `$path` — 実行パスが入っている

実行パスとは「コマンドとして扱うプログラムが入っているディレクトリのリスト」です。シェルはコマンドが打ち込まれると、実行パス中のディレクトリを順番に取り出し、そのディレクトリに「コマンドと同じ名前で」「実行可能な」ファイルがあるか調べます。そしてそのようなファイルがあったら、それを実行させます (図 2)。つまり Unix では、コマンドとは単なる「実行するプログラム」なのです。コンピュータでやることはすべてプログラムによって行うので、この考え方はとても合理的だと思います。

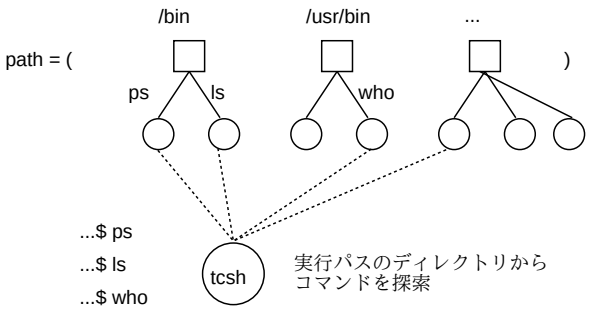

図 2: シェルによるコマンド探索

sol のユーザは管理者が設定した実行パスが最初から有効になっているため、標準のコマンドが利用できるわけです (試しに「`echo $path`」を実行してみるとよい)。そして、自分独自にコマンドを増やしたい場合は、たとえば次のようにすればできます。

```
...$ mkdir $home/bin ←ホームの下にディレクトリ bin 作る
...$ echo 'main() { puts("hello."); }' >test.c
...$ gcc -o $home/bin/hello test.c  ← hello 用意
...$ set path = ($home/bin $path)   ←パスに bin 追加
...$ rehash  ←パスやその内容を変更したことを tcsh に通知
...$ hello   ←新しいコマンドが
hello.       ←動くようになっている
```

自分のコマンドを入れておくディレクトリを毎回パスに追加したければ、後で出て来る設定ファイル `.tcshrc` のどこかに「`set path = ($home/bin $path)`」を入れておけば、いつでもここに置いた実行可能なファイルが使えるようになります。

4.3 ファイル名展開

ここまでで変数展開について説明しましたが、それと類似した機能である**ファイル名展開** (filename expansiton) についても説明しておきます。これまで現在位置にあるファイルの一覧を表示するのには `ls` を使って来たと思いますが、それ以外に次の方法もあります。

```
...$ echo *
CL16 Desktop IED_HOME WWW WindowsEdu a.out bin public_html
t1 t3 t4 test.c test1.txt test2.txt
```

これは何でしょう? 実は、コマンド行に次のようなパターンを書くとシェルはそのパターンにあてはまるファイル名を探して来て置き換えてくれるのです。

- `*` — 任意の文字の並び (ただし先頭の「.」は除く)
- `[...]` — かぎかっこ内に書いた文字のうちいずれか (正規表現と同じ)。

これを使うことで、「すべての C プログラムを消す」であれば「`rm *.c`」、「a〜d いずれかで始まるファイルだけ消す」であれば「`rm [a-d]*`」のように一気に指定できるわけです。

そのほかに普段使う展開として次のものもあります。

- ホームディレクトリの展開 — 「`~`」は自分のホームディレクトリに、「`~`ユーザ名」は指定したユーザのホームディレクトリに展開される。
- 分配法則 — 「`x{a,b,c}`」→ `xa xb xc` のように隣接する文字列とくっつけた状態で展開。たとえば「`.txt` と `.c` と `.s` のファイルを全部消す」だと「`rm *.{txt,c,s}`」のように書ける。
- コマンド置換 — 「‘コマンド…‘」はコマンドを実行してその出力結果に置き換えられる。見ても分かりにくいけれど「‘」はバッククォート文字なので「バッククォート置換」とも呼ぶ。

最後のコマンド置換はわりあい便利で、たとえば「`rm *.c`」で全部消すのではまずくて残したいファイルがある場合は、次のようにすればよいわけです。

(1) 「`echo *.c >t1`」で候補のファイル名をファイル `t1` に入れる。

(2) テキストエディタで `t1` を開いて消したらまずいファイルの名前は削除。

(3) 「`rm ‘cat t1‘`」で `t1` に入っている名前のものだけ削除。

なお、ここまでに「特別な働きを持つ文字」として`$*[]’‘~\`などが出てきました。このような文字を**メタ文字** (meta character) と呼びます。メタ文字を普通の文字として打ち込みたいときは、(1)「’...’」で囲むか (2) 直前に「\」を置くことで、特別な働きを止めることができます。

4.4 シェルスクリプト

ここまでに見てきたように、シェルのコマンドを使うとかなり複雑なことができます。そういう複雑なことを定期的にやる場合に、毎回考えるのは大変ですし、打ち間違えると面倒そうです。ではどうすればいいかというと、「ファイルに入れておけば」いいですね。

たとえば、自分のプロセスを観察するのに「`ps -u ka002689`」と毎回打つのは面倒ですから、次の内容を「`psme`」というファイルに入れておくことにします。

```
#!/bin/tcsh
ps -u $user
```

1 行目は「このコマンドは `tcsh` 用」というつもりですが、シェルは「`#`」ではじまる行は無視するので人間の覚え書き (コメント) の効果しかありません。2 行目が本体です。ユーザ名を固定してしまうと自分にしか使えませんが、シェル変数`$user` を書いておけば、これを動かした人のユーザ名に展開されますから便利ですね。そして、それを動かすには次のように `tcsh` に与えればいいです。

```
...$ tcsh psme  ←リダイレクションで tcsh <psme でも OK
   PID TTY          TIME CMD
 52964 ?        00:00:00 sshd
 52965 pts/202  00:00:00 tcsh
 76133 pts/202  00:00:00 ps
```

こうしておけば、もっと長いコマンドや、数行にわたるコマンドも、覚えておかなくてすみます。このように、シェルのコマンドをファイルに入れておいて実行させるものを**シェルスクリプト** (shell script) と呼びます (スクリプトとは台本という意味)。

これまで実行可能なファイルは C などの言語で書いて作っていましたが、実はシェルスクリプトも実行可能にできます。

```
...$ chmod a+x psme  ←誰でも実行可能にする
...$ ./psme          ←パス名指定して実行
   PID TTY          TIME CMD
 52964 ?        00:00:00 sshd
 52965 pts/202  00:00:00 tcsh
118550 pts/202  00:00:00 psme ←確かに実行中
118567 pts/202  00:00:00 ps
...$ mv psme $home/bin  ←自分の実行パスに入れると
...$ rehash             ←パスの内容変更したら rehash
...$ psme               ←コマンドとしてどこでも実行可能
(実行結果はおなじなので略)
```

そしてこれらの形で実行するときは、1 行目の「#!/bin/tcsh」は「このスクリプトは tcsh で実行してね」という指定として意味を持ちます。そのために入れておいたわけです。

なお、tcsh は各ユーザによって起動されたとき、$home/.tcshrc の内容を読み取ってスクリプトとして実行します。なので、ここに書いておくことで毎回実行したい設定を自動的におこなわせることができます。たとえば、シェル変数 prompt はプロンプト文字列なので、これを自分の好みに変更することなどはおすすめです。

4.5 スクリプトの引数と変数

シェルスクリプトによってコマンドが作れるとしても、それに対してオプションや引数を渡せなければあまり面白くはありません。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ます。従って、それらを参照したスクリプトを書くことにより、引数の値を活用できます。たとえば「ls -F -l ファイル…」をよく使うとしましょう。次のようなシェルスクリプトをコマンドにしたらどうでしょうか。

```
...$ cat $bin/lsf
#!/bin/tcsh
ls -F -l $1
...$ lsf t1 t2
-rw-r---w- 1 ka002689 faculty 39395 Jun 29 16:45 t1
```

あれ? ファイルを 2 つ指定したのに 1 つしか表示されない…のは当然で、スクリプト内部で引数$1 だけしか使っていませんね。ここは

```
#!/bin/tcsh
ls -F -l $1 $2 $3 $4 $5 $6 $7 $8 $9
```

でもいいのですが、こうやって並べるかわりに「$*」と書くことですべての引数をそこに埋め込むことができます。

さらに、ここでは説明しませんでしたが、シェルスクリプトの中で繰り返しや枝分かれや計算を行うこともでき、これらを組み合わせることで複雑な処理を行うプログラムを書くことも十分可能になっています。

演習 3 expr というのは、数式を与えるとその結果を計算してくれるコマンドである。たとえば「expr 1 + 3」は「4」を出力する。これを踏まえて、つぎのようなスクリプトを作ってみよ。

a. 2 つ引数を渡すと、その和を出力してくれる。
b. 任意個数の引数を渡すと、その和を出力してくれる。[12]
c. (自由課題。これは expr とは関係しない) 自分のコマンド用ディレクトリを作り、実行パスに入れなさい。その上で自分があると便利だと思うコマンドを考え、シェルスクリプトとして製作しなさい。

本日の課題 7A

本日の課題は「演習 1」「演習 2」「演習 3」に含まれる小問 (合計で 9 個) の中から 1 つ以上を選択し、結果をレポートとして報告して頂くことです。LMS の「レポート # 7」の入力欄に直接入力してください (別途作成したものをコピーペーストで貼っても構いません)。以下の内容がこの順に含まれるようにしてください。

- 冒頭に題名「コンピュータリテラシレポート # 7」、学籍番号、氏名、提出日付を書く。(グループでやったものはグループのメンバー全員の氏名も別途書く。)
- 課題の再掲を書く (どんな課題であるかをレポートを読む人が分かる程度に要約する)。
- レポート本体の内容 (やったこととその結果) を書く。
- 考察 (課題をやった結果自分が新たに分かったことや考えたこと) を書く。
- 以下のアンケートに対する回答。

  Q1. さまざまなフィルタやその機能について、どれくらい知っていましたか。新たに知って面白かったことは何ですか。
  Q2. シェルの様々な機能やシェルスクリプトについてどう思いましたか。
  Q3. リフレクション (今回の課題で分かったこと)・感想・要望をどうぞ。

なお、課題はグループでやって構いません。その場合も、(メンバー氏名を明記した上で) レポートは必ず各自で執筆してください。レポート文面が同一 (コピー) と認められた場合は同一であると認めた全員について点数にペナルティを科すことがあります。

[12] ヒント: スクリプトの中で「echo $*」を使うとすべての引数の間に空白 1 個がはさまった文字列が得られる。その空白をすべて「' + '」に置き換えると、expr に適する形になる。