第5章　フィルタとユティリティ

フィルタというのは、一般には空気や水などを透過させてほこりや不純物をろ過するものをいう。Unix のフィルタはその計算機版 (?) で、標準入力からテキストを読みとり、何らかの加工をして、結果を標準出力に出すものをいう。単純なフィルタを複数組み合わせて多用な処理をこなせることは Unix の特徴の 1 つである。本章では前半でフィルタを中心とした Unix のユテリティについて説明し、後半では「できあいの」フィルタにないような処理を簡単に作れる言語 awk について説明する。

5.1　Unix におけるユティリティの思想

5.1.1　「大きな」ユティリティと「小さな」ユティリティ

「ユティリティ」というのは、現実世界ではおおむねキッチンなどの横にあって洗濯やアイロン掛けなどができるようなスペースを指すようだが、計算機の世界では「ファイルの形式変換など汎用的な操作をやってくれる便利なプログラム」といった程度の意味で用いられる。

そして、「小さな政府」「大きな政府」という概念があるのと同様に、ユティリティにも「大きなユティリティ」と「小さなユティリティ」がある。大きなユティリティというのは、

- 1 つのユティリティプログラムに、沢山の機能がついていて、何でもできてしまうことをめざすもの

をいう。それ 1 つで何でもできるというのは便利そうではあるが、その代わり次のような弱点がある。

- どの機能を使うかといった指定が沢山必要で、使い方が複雑になりがちである。
- どれか 1 つの機能だけ使いたいときでも全機能を備えたプログラムが動くので遅くなりがちだし計算機資源の無駄づかいである。
- 機能をちょっと増やすとか訂正するといったことは、その巨大なプログラムを直さなければならず面倒だし実際上不可能なこともある。

これに対して、小さなユティリティというのは

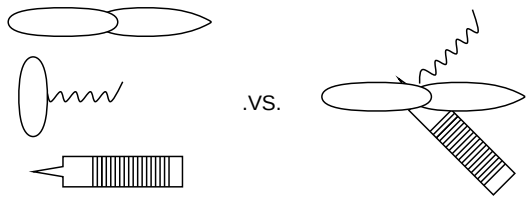

図 5.1: 単機能の道具と多機能の道具

- 1 つのユティリティプログラムは 1 つの (単純な) 機能しか備えていない

ものをいう。それではあんまり使いでがなさそうだと思うかもしれないが、複雑なことをやりたい場合には小さなユティリティを複数組み合わせて使えばできるように設計するわけである。こちらの利点は前の裏返しで、

- 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。

といったことである。その代わり、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、またそれに伴うオーバヘッドもいくらかはある。

5.1.2　データの受け渡し

Unix の場合、パイプライン (あるプログラムの出力を別のプログラムの入力に接続する) 機構がこの「仕組み」に相当する。ただし、これがうまく行くためには、どのプログラムの出力も別のプログラムの入力として役立つようになっている必要がある。例えば 1-2-3 形式のファイルは一太郎では読めない、とかいったことをやっていては駄目なわけである。

では、計算機の世界で汎用的に使えるデータ形式とはどんなものだろう? 色々な考え方があり得るが、Unix の場合それは「テキストファイル」つまり人間が読み書きできるようなファイルなら何でも、という方針を取っている。そうしておけば、データを人間が用意するのも楽だし (エディタで打ち込めばよい)、途中結果もそのまま画面に表示して調べることができる。

例えば、次のようなファイルを考えてみよう。

```
1994.4.13       kuno    s       3200    floppy
1994.2.20       kuno    s       4000    notebook
1994.4.20       ohki    b       99000   hard-disk
1993.12.20      terano  b       240000  mac-av880
```

```
1994.5.12       terano  s       2000    card-folder
1994.3.18       kuno    b       82000   display
1993.11.10      ohki    s       4000    notebook
1994.5.13       terano  s       1600    floppy
```

まあ、意味は見れば明らかですね? これについて、次のような処理をしたいとする。

a. 日付の区切りを「.」でなく「/」に取り換えたい。

b. 金額を右そろえしたい。

c. 消耗品 (分類が「s」のもの) だけ取り出したい。

d. 品物のアルファベット順や金額の高い順や日付順に並べたい。

e. 各人が何品ずつ買ったか調べたい。

f. 品名と金額のみのリストにしたい。

g. 金額の合計や平均を求めたい。

h. 個人ごとにいくら買ったか集計したい。

どうしますか? プログラムを書く? 今回説明するユティリティを使えば、すべてプログラムは書かず、ユティリティの組合せで処理できる。

「定型業務」と「非定型業務」という言葉を聞いたことがおありだろうか? 同じ処理を何回も何回もやる「定型業務」なら、そのためにプログラムを開発しても引き合うかも知れないが、ちょっとこれこれのデータを組み合わせてレポートの材料にする、といった「非定型業務」ではそのためにプログラムを書くのはえらく損である。おそらく、皆様が修論などのためにやるデータ整理はほとんどこちらになるだろう。だから、以下の内容をマスターして自在にデータ加工ができるようになっておいて損はないですよ、と思うわけである。

5.1.3 漢字はどうする?

ここで漢字がらみの注意を 1 つ。漢字の入ったファイルを正しく処理するには、次のことをやってください。

- 漢字対応のフィルタ群は smb のみにあるので、smb で作業する。

  「`export LC_CTYPE; LC_CTYPE=japanese`」を実行して、フィルタ群の動作を標準 (英語モード) から日本語処理モードに切替えておく。

- 処理する日本語ファイルは EUC コードによること。つまり、

  `nkf -e` 入力ファイル `|` 加工… `|` `nkf >`出力ファイル

などのようにして使う必要がある。

- コマンドの引数に漢字を指定する場合には、それも EUC でなければならない。だから、いきなり打ち込むのではなくシェルスクリプトにして、EUC コードに変換してから実行させる。ついでに上の export 指定も一緒にスクリプトに入れておくとよい。

これらを守れば、漢字のものもちゃんと処理できます。以下ではいちいち漢字を入れるのが面倒なので全部英字でやります。

5.2　Unix のフィルタ

5.2.1　expand — タブの展開

Unix のテキストファイルには、タブ文字が使われることが多い。タブ文字とは「次のタブ位置 (標準では 8 の倍数の欄) まで移動する」機能を表す制御文字である。これを使えば表のようなものが比較的容易に作れる。

しかし用途によってはテキストファイルにタブが含まれない方がいいこともある。例えばファイルの各行から特定の欄位置だけ抜きだそうと思った場合、タブが入っていなければ「n 文字目から m 文字目まで取り出す」ことでできる。このため、タブ文字を適当な個数の空白に置き換えることで取り除くのが expand である。その使い方は簡単で、

```
expand                  --- 標準入力を展開
expand ファイル …      --- 指定したファイル (群) を展開
```

となっている。このように、Unix のフィルタの基本型は「入力ファイルが引数として指定されていればそのファイルから読み、そうでなければ標準入力から読む」という形になっている。(ただし tr だけは引数としてファイルが指定できないので入力リダイレクトを使う。)

5.2.2　nkf — 漢字コード変換

前に説明した nkf も expand と同様、ファイルが指定されていなければ標準入力から読むフィルタである。

```
nkf ファイル …       --- JIS への変換
nkf -e ファイル …    --- EUC への変換
nkf -s ファイル …    --- SJIS への変換
```

あと、expand は欄位置をバイト数で数えるだけだから、漢字が入っている場合には JIS のようにバイト数と欄位置が一致しないのを展開するとうまく行かない。EUC に変換してから expand すると大丈夫。

5.2.3 fold — 折り畳み

Unix ではテキストファイルの行はどんなに長くても「基本的には」よい。しかし実際には行単位で処理をするツールもあるから、長すぎる行はうまく処理できないことがある。そこで指定した長さより長い行があったら改行を入れて短くするのが `fold` である。

```
fold ファイル …              --- 72 文字で折り畳む
fold -文字数 ファイル …   --- 指定した文字数で折り畳む
```

なお、漢字コードが含まれているファイルの場合にはその 1 バイト目と 2 バイト目の途中で改行されるとおかしくなるので、`fold` でなく `jfold` を使ってください。

5.2.4 head と tail

入力のなかの先頭部分だけ、またはおしまいの部分だけ取り出したいというのもよくあることである。(たとえばサンプルとして最初の方だけ見たい、処理記録の末尾だけ見てエラーで終わっていないか調べるなど。) それには head と tail が役に立つ。

```
head -行数     : 先頭から指定した行数だけ出力
tail -行数     : 最後の n 行だけ出力
tail +行数     : n 行目以降のみ出力
```

ところで、これらを組み合わせると「n 行目から m 行だけ出力」になる。

```
... | tail +n | head -m | ...
```

5.2.5 tr — 文字置換

Tr は既に前にも出てきたが、一般には入力の各文字を別の文字で置き換えるようなフィルタである。例えば次の通り。

```
% echo 'This is a pen.' | tr ' ' '*'
This*is*a*pen.
```

置き換えの組は複数いっぺんに指定できる。つまり 1 番目の引数と 2 番目の引数の対応する文字がそれぞれ置き代わる。

```
% echo 'This is a pen.' | tr ' aiueo' '*$#%@!'
Th#s*#s*$*p@n.
```

だから、小文字を大文字にするには「tr abcdefghijklmnopqrstuvwxyz ABCDEFGHIJKLMNOPQRSTUVWX
とすればいいのだが、これでは余りにも打つのが大変なので「…から…まで」という書き方を許している。それが「tr a-z A-Z」だったわけである。

なお、2 番目の引数が短い場合にはその最後の文字が繰り返し指定されたと見なされる。

```
% echo 'This is a pen.' | tr ' a-z' '*@'
T@@@*@@*@*@@@.
```

「置き換える」代わりに「消してしまう」こともできる。その場合には 1 番目の引数だけ与えて-d というオプションを指定する。

```
% echo 'This is a pen.' | tr -d ' .'
Thisisapen
```

これは不要な制御コードを消してしまうのに役にたつ。制御コードは直接キーボードから入れられないので、\nnn という記法でその文字コードを 8 進数 3 桁で指定する。例えば「tr -d '\000-\011\013-\037'」で改行以外の制御コードを全部消してしまうことができる。もっとも、消す場合には「〜以外を消す」といいたい場合も多い。その場合には-c というオプションを指定する。

```
% echo 'This is a pen.' | tr -cd 'aiueo'
iiae
```

このオプションは-d と一緒でなくても使える。

```
% echo 'This is a pen.' | tr -c 'aiueo' '*'
**i**i*a**e**
```

ここで「連続した*は 1 個にまとめたい」場合にはさらに-s を指定する。

```
% echo 'This is a pen.' | tr -cs 'aiueo' '*'
*i*i*a*e*
```

tr のよくある応用の 1 つは、英単語を 1 行ずつに切り分けることである。その場合は*の代わりに改行を使えばいいわけである。

```
% echo 'This is a pen.' | tr -cs 'A-Za-z' '\012'
This
is
a
pen
```

5.2.6 grep 属 — パターン探索

入力ファイルの中から特定のパターンを含む行だけを取り出してくれるツールのことを Unix では伝統的に grep [1] と呼んでいる。そして、「属」という言葉の通り用途に応じて次の 3 つを使い分けるようになっている。

```
fgrep パターン ファイル …
grep パターン ファイル …
egrep パターン ファイル …
```

これらはいずれも、「パターン」として何が書けるかが違っているだけである。これらの中で一番単純なのは fgrep で、パターンとして単なる文字列だけが書ける。実はこれでも十分な場合も多い。たとえば自分が「that」を「taht」とミスタイプする傾向があるとしたら、そのミスタイプの行を探すのには

```
fgrep 'taht' ron.txt
```

のようにして使うのは結構役に立つ。しかしこれだと「That」を「Taht」と間違えたのはもう捜せない。その場合には grep が役に立つ。つまり

```
grep '[Tt]aht' ron.txt
```

とすれば先頭が大文字のも小文字のもまとめて捜せる。しかしこれと一緒に「this」を「tihs」にしてしまうのも探そうと思うと grep でも役不足で egrep が必要になる。egrep であれば「かっこ」と「または」が使え

```
grep '[Tt](aht|ihs)' ron.txt
```

のように書くことができる。

以上がイントロで、具体的な grep と egrep のパターンについて次にまとめておく。まず両者に共通のもの。

```
c       : c という文字そのもの (c は以下の特殊文字でないこと)
\c      : c という文字そのもの (c は以下の特殊文字)
[…]     : …の文字のうちのどれか
[^…]    : …の文字の以外の文字のどれか
.       : 任意の 1 文字
X*      : X が 0 個以上
X\{m\}  : X がちょうど m 回
^α      : 行の先頭にある α
α$      : 行の末尾にある α
```

次に egrep のみのもの。

[1] Grobal look for Regular Expression and Print の略

(α)　: α と同じ (くくりだしのかっこ)
α|β　: α または β

一方で、grep にしかないものもある。

\(α\) : α と同じだが、ただしマッチした部分を覚える
\n　　: 覚えたものの n 番目

たとえば/usr/dict/words というファイルには多数の英単語が入れてあるが、その中から様々な単語を取り出す例を挙げておく。

```
(tion|tive)$      --- 末尾が「tion」、「tive」で終わる単語
^............$    --- ちょうど 12 文字の単語
^\(.\)\(.\).\2\1$ --- 5 文字の回文 (?)
```

5.2.7　sed — 文字列置換

tr が 1 文字単位での置き換えだったのに対し、文字列単位での置き換えをしたい場合には sed を使う。すなわち、

sed 's/α/β/' ファイル …

はファイルの各行について、もしパターン α があったらそれを β で置き換えることを行う。置き換えは 1 行について 1 回しか起こらないが、もし何回でも可能な限り起こってほしければ

sed 's/α/β/g' ファイル …

とする。なお、α の中には上の grep と同じパターンが使える。そして、β のでは&と書くと α にマッチした文字列全体を意味し、また\1 などもパターン側と同様に使える。例えば次の通り。

```
% cat t.txt
This is a pen.
That is a fish.
% sed 's/is/IS/g' t.txt
ThIS IS a pen.
That IS a fISh.
% sed 's/[^A-Za-z]is[^A-Za-z]/IS/g' t.txt
ThisISa pen.
ThatISa fish.
% sed 's/\([^A-Za-z]\)is\([^A-Za-z]\)/\1IS\2/g' t.txt
This IS a pen.
That IS a fish.
```

なお、sed には「s」以外にも多数の指令があるのだけれど、それらを駆使するよりは後述の awk などを使った方が素直で分かりやすいので説明しない。興味がある人は「sed & awk」といった本を参照のこと。

5.2.8 sort と uniq

計算機の世界では、データを何らかの決まった順番に並べかえることをソートする、という。(昔は分類と訳していたが最近はさすがに整列と訳すのがふつう。) sort の一番簡単な形は

```
sort ファイル …
```

で、これだと入力を各行ごとに全体として文字コード順になるように並べる。もう少し芸を細かくして

```
sort -fnr ファイル …
```

とすると f、n、r の有無によって並べ方を多少制御する:

```
f  --- 大文字小文字の区別をしない (通常はする)
n  --- 数値として比較する (通常は文字列として比較)
r  --- 大きい順に並べる (通常は小さい順に並べる)
```

この必要性は次を見ればわかる。

さらに芸を細かくすると各行が (互いに空白で区切られた) 複数の欄からなっていて、そのうちのどの欄についてどう並べるかを指定できる。それには

```
sort +番号 fnr +番号 fnr … ファイル …
```

による。ここでそれぞれの番号は「何番目 (最初の欄を 0 とする) の欄に基づいて整列」を意味し、先に書かれたものほど優先される。そして「fnr」の部分はあってもなくてもよく、あった場合には先と同じ意味になる。

さて、パラメタなしの sort で並べると、同じ内容の行は隣接して並ぶことになる。

```
% tr -cs 'A-Za-z' '\012' <t.txt | sort
That
This
a
a
fish
is
is
pen
```

多くの場合、同じものが何回もでてきてもうるさいだけなので、隣接する同じ行を取り除いてそれぞれ 1 回ずつしか現れないようにしたいことが多い。これを行うのが uniq である。

```
% tr -cs 'A-Za-z' '\012' <t.txt | sort | uniq
That
This
a
fish
is
pen
```

これで「単語帳」ができることになる。しかしどの単語が何回現れたか知りたいかも知れない。その場合には uniq に-c オプションをつけると何回重複していたかを数えてくれるようになる。

```
% tr -cs 'A-Za-z' '\012' <t.txt | sort | uniq  -c
   1 That
   1 This
   2 a
   1 fish
   2 is
   1 pen
```

ここまでくるとついでに、回数の多い順に並べたくなるかも知れない。その場合にはもう 1 回 sort すればよい。

```
% tr -cs 'A-Za-z' '\012' <t.txt | sort | uniq  -c | sort +0nr
   2 a
   2 is
   1 That
   1 This
   1 fish
   1 pen
```

ここでは「r」がついているから多く使われている単語から順に見えるが、「r」なしにすれば少ないものから順に見えるので、打ち間違いを発見するのに役に立つだろう。

5.2.9　wc – 行数、語数、文字数を数える

wc はほとんど説明の必要がなく、ただ単に

```
wc ファイル …
```

により入力ファイルの行数、語数、文字数を表示するだけである。なお、オプションとして-l、-w、-c を指定するとそれぞれ行数、語数、文字数のみを表示するようになる。例えばさっきの例で具体的な単語には興味がなく、ただ何種類の単語が使われていたかだけ調べるのには次のようにする。

```
% tr -cs 'A-Za-z' '\012' | sort | uniq | wc -l
6
```

5.3 Awk

5.3.1 Awk とは

これまでに見てきたように、Unix のテキスト用フィルタとシェルスクリプトを組み合わせることで様々なテキスト処理を行うことができる。しかしそのやり方は結構パズルみたいだし、内容によってはいちいち多数のプロセスを組み合わせるため効率が悪いかも知れない。そこで、テキスト処理を記述するための小さな言語 awk(「オーク」と読む) が登場する。これを使えば、多くのテキスト処理が簡単に効率よく記述できる (なお、awk 言語には簡潔な旧版と多くの機能を持つ新版とがある。ここでは簡単に説明できる旧版を取り上げている)。

awk はいちばん簡単な使い方においては、sed の親類のようなものだと考えることができる。つまり、

```
awk '{ プログラム }' ファイル …
```

により、指定したファイル (群) の各行に対して「プログラム」による処理を行うことができる。sed と同様、ファイルを指定しないと標準入力を処理してくれるのでフィルタとしても使える。ただし、sed と違ってプログラムの中で print(ないし printf) によって明示的に打ち出したものだけが出力に現われる。

もう 1 つの特徴は、awk では入力の各行を空白部分 (スペースまたはタブ文字のならび) で区切られた複数の欄に自動的に分けてくれることである。各欄は$1、$2、$3…という名前で参照できる。例えば ps では最初の欄が PID、5 番目の欄がコマンドだからこれらを抜き出すのは次のようにすればよい。

```
% ps
   PID TT       S  TIME COMMAND
 19061 pts/7    O  0:00 ps
 18789 pts/7    S  0:03 bash
% ps | awk '{ print $5, " : ", $1; }'
```

```
COMMAND  :  PID
awk  :  19051
bash  :  18789
ps  :  19050
```

print は見ての通り引数を順に出力する。(そして最後に改行する。) 出力の位置を縦に揃えたい時には代わりに printf を使う。

```
% ps | awk '{ printf "%12s : %8s\n", $5, $1; }'
     COMMAND :      PID
          ps :    19070
        bash :    18789
         awk :    19071
```

printf では出力は最初の引数 (書式文字列) によって制御される。基本的に書式文字列はそのまま出力されるが、ただしその中に「%ns」というものがあると、幅 n の欄が取られてそこに図 x にあるように対応する引数が右づめで埋め込まれる。左づめにしたい場合には次のように $-n$ にすればよい。

```
% ps | awk '{ printf "%-12s %8s\n", $5, $1; }'
COMMAND           PID
ps              19074
bash            18789
awk             19075
```

この手のプログラムだけでも、ファイルの中の特定欄だけを抜き出したり、欄の順序を入れ換えたり、幅を揃えて整えるなどの役に十分立つ。

5.3.2　Awk スクリプト

ここまでの例では「プログラム」の中身が文 1 つだけだったが、複数の文を書いてもよいし、文として if 文や while 文なども使える。そうなるととても 1 行では書ききれないし、毎回打ち込むよりファイルに入れておいて繰り返し使うほうがよい。すなわち「awk スクリプト」にするわけである。それにはプログラムをファイルに入れておき

```
awk -f ファイル名
```

により動かせばよい。そしてもう 1 つの方法はインタプリタ指定を使うことで、例えば

```
#!/usr/bin/awk -f
{ num += 1; printf "%6s: %-s\n", num, $0; }
```

(なお、$0 というのは入力行全体が入る変数である。) これを number というファイルに入れて実行可能にし、自分のコマンドディレクトリに置いておけば

```
% cat number
#!/usr/bin/awk -f
{ num += 1; printf "%6s: %-s\n", num, $0; }
% number t41.c
     1: /* t41.c -- print command arguments. */
     2:
     3: main(int argc, char *argv[]) {
     4:   int i = 0;
     5:   while(i < argc) {
     6:     printf("%d: %s\n", i, argv[i]); i = i + 1; }
     7: }
%
```

のように普通のコマンドぽく使うことができる。

5.3.3 Awk の制御構造

ここまででではずっと 1 行しかない awk プログラムをやってきたが、実際には awk のプログラムは次のような一般的構造をしている。

```
パターン { 文; … ; 文; }
パターン { 文; … ; 文; }
…
パターン { 文; … ; 文; }
```

ここでパターンは「/正規表現/」のようなもの、または BEGIN か END である。BEGIN に続く文はプログラムの最初に 1 回実行され、END に続く文はプログラムの最後に 1 回実行される。そして、それ以外の文は、その行が正規表現にマッチする時に実行される。パターンを書かなかった場合はすべての行について実行される (これまでの例はこれだったわけである)。

では、BEGIN や END の機能は何のためにあるのだろう? それはつまり、初期設定やあと始末ということになる。例えば、1 行に 1 個ずつ数値が入っているファイルがあるとして、その数値の合計を求めてみる。

```
% cat sum1
#!/usr/bin/awk -f
BEGIN { sum = 0; }
      { sum += $1; }
END   { printf "sum = %s\n", sum; }
```

```
% cat t.data
  10.5
  20.2
  18.18
% sum1 t.data
sum = 48.88
%
```

実は初期設定していない変数は 0 かつ空文字列であると見なされるので、BEGIN の行はなくても同じである。なお、awk は他のフィルタと同様、入力ファイル名が指定されていればそちらを読み、指定されていない時は標準入力を読む (ので、上の実行例でリダイレクトは不要なわけである)。

5.3.4　Awk の文

ここまでに出てきた文は代入文と print 文と printf 文だけだったが、これを含めて awk には次のような文がある。

```
変数 = 式;                  ←代入文 (+=、-=なども可)
print 式, ...;              ←単純な出力、改行つき
printf 書式文字列, 式, ...;  ← C の printf と同様
if(条件) 文 [ else 文 ]     ← if 文
while(条件) 文              ← while 文
for(式; 条件; 式) 文        ← C の for 文と同様
for(変数 in 配列) 文        ←配列用 (後述)
break;                      ←ループの抜け出し
continue;                   ←ループの続行
{ 文 … }                    ←複合文
```

例えば 1 行のなかに沢山の語が入っているのを全部 1 行ずつに分けるには次のようにすればよい。

```
% cat sepr
#!/usr/bin/awk -f
{ for(i = 1; i <= NF; ++i) print $i; }
% cat t1.data
  10.5 20.2 18.3
  18.18 5
% sepr t1.data
10.5
20.2
18.3
```

```
18.18
5
%
```

なお、変数 NF には現在処理中の行に含まれている欄の数が入っている。同様に、変数 NR には現在処理中の行が何行目かが入っている。

5.3.5 Awk の配列

awk にも普通の言語と同様配列の機能がある。例えば次の例は入力ファイルを上下逆転させる。

```
% cat rev
#!/usr/bin/awk -f
    { line[NR] = $0; }
END { for(i = NR; i > 0; --i) print line[i]; }
% cat t3
What
is
this?
% rev t3
this?
is
What
%
```

ところで、普通の言語と違って awk では「配列の添字は数でなくてもいい」という特徴がある。例えば文字列を添字にすることもできる。このような機能を「連想配列」と呼び、とても便利に使うことができる。例えば皆様のレポートを集計するのに「a は 7 点、b+は 6 点、…」といった点数の置き換えをやりたいとする。

```
% cat map
#!/usr/bin/awk -f
BEGIN { pt["a"] = 7; pt["b+"] = 6; pt["b"] = 5; }
      { if(pt[$2] == "") pt[$2] = "?";
        printf "%-8s %s\n", $1, pt[$2]; }
% cat t2.data
kuno  c
ohki   a
terano b+
% map t2.data
```

```
kuno     ?
ohki     7
terano   6
%
```

まだ「何も入れてない」要素には空文字列が入っているので、そのような要素を参照しそうになった時は (たぶんデータの打ち間違いだろうから) "?" という印を入れてそれとわかるようにしている。

連想配列のもう 1 つの有用な使い方は名前ごとの集計である。たとえば、上述のような処理をした後で、誰が合計何点かを計算したいとする。連想配列に合計を取って行けばごく簡単である…が、集計し終わったあと「誰と誰がいたか?」を知るにはどうしたらよいだろう?

```
% cat accum
#!/usr/bin/awk -f
    { acc[$1] += $2; }
END { for(name in acc)
        printf "%-8s %4s\n", name, acc[name]; }
% cat t3.data
kuno 3
ohki 5
terano 3
kuno 1
terano 7
% accum t3.data
ohki        5
kuno        4
terano     10
%
```

このように、for 文の 2 番目の形を使うと「連想配列の添字を 1 個ずつ変数に取り出しながらループする」ことができる。ところで、取り出される順番は「でたらめ」で制御できない。だから名前順や点数順にしたければ… その時は sort を使う! つまり、awk は他のフィルタと組み合わせて使う「ちょっとした処理のための」言語であって、それ自体で何でもやってしまおうというものではないのである。

5.3.6　Awk の関数と式と条件

最後になったが、awk ではちょっとした処理に便利ないくつかの組み込み関数が用意されている。

```
int(数値)              ←少数点以下切捨て
exp(数値)              ←指数関数 (他に sqrt、log もある)
length(文字列)         ←文字列の長さ
substr(文字列, i, j)   ←文字列の i 文字目から j 文字取り出す
sprintf(書式文字列, 式, …)  ← printf と同様だが文字列を返す
```

既に説明せずに出てきたが、式の中では C と同様の演算 (加減乗除、インクリメント、デクリメント) が使える。また、条件としては

```
式 == 式          ←他に!=、<、>、<=、>=
式 ~ パターン     ←式の中にパターンが含まれる
式 !~ パターン    ←式の中にパターンが含まれない
```

などが使える。

5.4 おわりに

おしまいに、5.1.2 節で挙げた「例えば…」をこれまで説明してきた道具でどうやるかを示しておこう。

a. 日付の区切りを「.」でなく「/」に取り換えたい → 単に「sed 's/./g'」でもよさそうだが、品名の中にピリオドがあると困るかも? だから

```
% sed 's/\./\//' test.data | sed 's/\./\//'
1994/4/13       kuno    s       3200    floppy
1994/2/20       kuno    s       4000    notebook
1994/4/20       ohki    b       99000   hard-disk
1993/12/20      terano  b       240000  mac-av880
1994/5/12       terano  s       2000    card-folder
1994/3/18       kuno    b       82000   display
1993/11/10      ohki    s       4000    notebook
1994/5/13       terano  s       1600    floppy
```

なお、「.」は任意の 1 文字をあらわすパターンだし、「/」は sed の s コマンドの区切りだからどちらも前に\をつけた。

b. 金額を右そろえしたい → 単に awk 読み込んで書き出す時にそろえを指定すればよい。

```
% awk '{printf "%-10s %-8s %-8s %8s %s\n",$1,$2,$3,$4,$5;}' test.data
1994.4.13  kuno     s            3200 floppy
1994.2.20  kuno     s            4000 notebook
1994.4.20  ohki     b           99000 hard-disk
```

```
1993.12.20 terano   b          240000 mac-av880
1994.5.12  terano   s            2000 card-folder
1994.3.18  kuno     b           82000 display
1993.11.10 ohki     s            4000 notebook
1994.5.13  terano   s            1600 floppy
```

c. 消耗品 (分類が「s」のもの) だけ取り出したい → grep でもいいけど、位置を数えるのが awk の方が簡単。

```
% awk '{ if($3 == "s") print $0; }' test.data
1994.4.13       kuno    s       3200    floppy
1994.2.20       kuno    s       4000    notebook
1994.5.12       terano  s       2000    card-folder
1993.11.10      ohki    s       4000    notebook
1994.5.13       terano  s       1600    floppy
```

d. 品物のアルファベット順や金額の高い順や日付順に並べたい→ 完全に sort むけの問題ですね。金額だと次の通り。

```
% sort +3nr test.data
1993.12.20      terano  b       240000  mac-av880
1994.4.20       ohki    b       99000   hard-disk
1994.3.18       kuno    b       82000   display
1993.11.10      ohki    s       4000    notebook
1994.2.20       kuno    s       4000    notebook
1994.4.13       kuno    s       3200    floppy
1994.5.12       terano  s       2000    card-folder
1994.5.13       terano  s       1600    floppy
```

e. 各人が何品ずつ買ったか調べたい → 名前を添字とした連想配列でもできますが、次のではいかが。

```
% awk '{ print $2; }' test.data | sort | uniq -c
   3 kuno
   2 ohki
   3 terano
```

f. 品名と金額のみのリストにしたい。→ awk で取り出した後、sort と uniq が必要。なぜかわかりますか?

```
% awk '{ printf "%-12s %8s\n", $5, $4; }' test.data | sort | uniq
card-folder      2000
```

```
display          82000
floppy            1600
floppy            3200
hard-disk        99000
mac-av880       240000
notebook          4000
```

g. 金額の合計や平均を求めたい → awk で集計ですね。

```
% awk '{sum+=$4;} END{printf "%9.1f %9.1f\n",sum,sum/NR;}' test.data
 435800.0   54475.0
```

h. 個人ごとにいくら買ったか集計したい → これは連想配列がいいでしょうね。名前の ABC 順にしたければ再度 sort します。

```
% awk '{m[$2]+=$4;} END{for(i in m)printf "%-8s %8s\n",i,m[i];}' test.data
ohki        103000
kuno         89200
terano      243600
```

5.5 演習

5-1. /usr/dict/words を材料にして、次の項目から 3 つ以上調べてみよ。(各種フィルタを組み合わせてやる)

a. 末尾が tion、tive で終わる単語それぞれの個数。
b. tion、tive を含むがこれらが末尾にない単語の数 (と代表例)。
c. 一番長い単語。[2]
d. 長さ何文字の単語はいくつあるかの表。[3]
e. 一番多くの母音を含む単語。[4]
f. 一番長い連続した母音列を含む単語。[5]

5-2. 次の作業から 1 つ以上選び awk でフィルタを書いてみよ。

a. `who` の出力に現われる日付は「May」「Jun」などと書いてあるため日本人にはなじみにくい。日付部分を自分の好みの形式に取り換えるフィルタを書け。

[2]ヒント: すべての文字をたとえば「@」にしてしまい、その後逆順に整列すれば先頭に一番長い行が来ます。
[3]ヒント: 前問のヒント通りにやった後 `uniq -c` を使う。
[4]ヒント: 母音のみを「@」、それ以外の普通の文字を削除した後で前々問のようにすれば、最も多くの母音というのは何個かわかる。あとは grep で探す。
[5]ヒント: 母音のみを「@」、それ以外をすべて改行文字にした後逆順ソートすると、最も長い母音列は何文字かわかる。あとは grep で探す。

b. `ls -l` の出力に現われるファイルモード表示や所有者表示などは取り除いてしまい、もっとコンパクトな出力に加工するフィルタを書け。(何を出力に含めるかは自分の好みでよい。)

c. `ls -l` の出力を加工して、ファイルの大きさの平均値を計算し出力するフィルタを書け。

d. `ps uax` の出力を加工して、誰が何個プロセスを動かしているか調べるフィルタを書け。

e. `du -a` のようにあまり各行が長くなく、行数が多いような出力は `ls` のように N 段組になってくれると嬉しい。それを行なうようなフィルタを書け。[6]

5-3. 次に挙げたコマンドのうちからあったら便利そうだと思うものを 1 つ以上選んで、シェルスクリプトとして用意し、自分のコマンド用ディレクトリに置け。ものによっては awk スクリプトを別のファイルとして用意した方がやりやすいかも知れない。

a. 自分の持っているファイルの大きい方からベスト 10 を表示する。10 というのをパラメタで指定できるとなおよい。[7]

b. 自分のプロセス全部をプロセス番号順に表示するコマンド。できればオプションによって大きさ順や CPU 消費順にできるとなおよい。

c. `find . -print` の出力は下左のように各行に重複が多くて見にくい。これを整理して、右のように字下げを活用して表示するコマンドを作れ。なお、find のままだと順番が ABC 順にならないので、ABC 順に並べること。[8]

```
./awk                 awk
./awk/number            number
./awk/sepr              sepr
./sam                 sam
./sam/t41.c             t41.c
./sam/times1            times1
./sam/word1             word1
./t1                  t1
```

[6] ヒント: 多分次のような順番になるだろう。まずファイルを順に配列に読み込む。次に、最も長い行と全部で何行あるかを調べる。画面の幅 (80 としてよい) を長い行の文字数で割り、1 段の幅 W を決め、段組み数 N を決める (自動的に 1 段の行数 H も決まる)。最後の段に入る行以外は、うしろに空白を連結して段の幅にそろえる。あとは 1 行目、H+1 行目、…、(N-1)H+1 行目を出力し、改行し、2 行目、H+2 行目、…、(N-1)H+2 行目を出力し、…を繰り返せばよい。

[7] ヒント: `du -a` と `sort +0nr` と `head` を使う?

[8] ヒント: たとえば、awk に食べさせる前に「/」を空白に置き換え、awk の中では欄数に応じて字下げの空白を出してから最後の欄のみ出力する。

今回は各問題の中に選択の余地がありますので、各課題とも報告してください。