本稿は Linux Japan 誌 2000 年 9 月号に掲載された記事に補筆修正したものです．

AWK でデータ処理

Linux を高級電子文具として使っていると，シェルスクリプトでは無理だけど，C でプログラミングするにはちょっと大袈裟と言うようなケースにまま遭遇します．筆者の日常に照らせば，四則演算や初等関数(sin,cos,log,exp) 程度を用いた実験のデータ処理などは、その典型です．そんな時，AWK がぴったりのツールとして大活躍します．機能が練られているので，覚えることも少く，手放せない逸品です．もちろん，大概の Linux 配付系で GNU の awk(gawk) が標準インストールされますから，誰でも簡単に使い始めることができます．

AWK の情報源

AWK の名前自身はその由来も含めて，既に皆さんご存じでしょう．あるいは，もう完璧に使いこなしているとおっしゃる読者の方もおられるでしょう．が，どんなに良く知っているつもりでも，改めてネットワーク上で探索すると，知らなかった事柄などを必ず発見するものです．その意味で，ネット上の FAQ 集は大変勉強になります [1]W3．筆者も，オンライン自習コース [2]W3 があるのを見つけて，感心しました．もちろんプログラミング言語としても歴史のある AWK のことですから，書籍も多く刊行されています．三人の偉大な開発者が自ら記した『プログラミング言語 AWK』[3] は必携の一冊ですし，UNIX 関連の書籍に関しては右に出るもののない O'Reilly からの『sed & awk』[4]W3 も定番でしょう．もちろん gawk のマニュアルあるいはユーザーガイド [5]W3 でもじゅうぶん学習可能です．

入門編

AWK はインタプリタ言語で，ちょこっと書いてすぐ実行できますから，堅苦しい文法解説よりも，実際に動かしてみるのが一番です．まずは，処理対象となるデータを準備しましょう．そうですね，/tmp のディレクトリ一覧なんてどうでしょうか．

```
$ ls -la /tmp > awk.data
```

基本型：パターン { アクション }

AWK は，まず入力行を 1 行ずつ読み込み ($0 として参照)，それらを区切り記号 (通常は空白) を基にフィールド ($n として参照) に分解します．そして，文字列照合や式などのパターンに適合するかどうか調べ，適合した場合には，指定されたアクションを起こす仕組みになっています．例えば

```
$ awk '/root/ { print $0 }' awk.data
```

は，awk.dat から一行ずつ読み込み，行内に文字列 root を含む場合，その行全体 ($0) を表示します．コマンドライン上でパターンとアクションを指示する場合，シェルに解釈されないようにシングルクォートで囲わなければなりません．スラッシュで囲まれた部分は sed でお馴染み**正規表現**となっています．この例は

```
$ awk '/root/ { print }' awk.data
$ awk '/root/' awk.data
```

と指示することもできます．print ... では引数省略時の既定が $0 となっており，アクションの省略時既定が `{ print $0 }` となっているからです．また，パターンの省略時既定は常に適合です．したがって

```
$ awk '{ print }' awk.data
```

は，“cat awk.data” と同じ結果となります．ただし，パターンとアクションの両方を省略すると，何もしません．正規表現への適合を調べる対象 (フィールドなど) を指示するにはチルダ記号 '~' を使います．

```
$ awk '$1 ~ /drw/' awk.data
$ awk '$3 ~ /ro/ {print $1,$2,$3}' awk.data
$ awk '$3 !~ /bi/ {print $(NF-1),$NF}' awk.data
$ awk '$1 ~ /^d/' awk.data
$ awk '$NF ~ /\/$/' awk.data
```

NF は処理中の行に含まれるフィールド数を保持している組み込み変数です．したがって，$NF は最後のフィールドの内容を表します．フィールドを省略した場合既定は $0 です．すなわち，行全体を調べます．上記の例のうち最後の 2 つは，正規表現としてメタキャラクタ ^ と$ を使っています．次の節を読めばその意味がわかります．

NF を使ったので，組み込み (built-in) 変数をまとめて表 1に示します．

表 1　awk の組み込み変数: 全て大文字!

変数名	機能と既定値
ARGC	コマンド行の引数の数+1
ARGV	コマンド行の引数の配列
FS	入力フィールド区切子（空白とタブ）
RS	入力レコード区切子（'\n'）
OFS	出力フィールド区切子（空白 ' '）
ORS	出力レコード区切子（'\n'）
OFMT	数字の出力フォーマット（%.6g）
NR	入力レコード総数
FNR	現在のファイルのレコード番号
NF	入力レコードのフィールド数
FILENAME	現在の入力ファイル名
SUBSEP	添字の区切子 ('\034')
RSTART	match 関数で適合した文字列の開始位置
RLENGTH	match 関数で適合した文字列長さ
ENVIRON	環境変数を保持する配列
CONVFMT	数字の出力フォーマット (POSIX 互換)
gawk における追加	
ARGIND, ERRNO, FILEDWIDTHS, IGNORECASE, RT	

正規表現 (Regular Expression)

パターンとなる正規表現は AWK に留まりません．sed や grep の延長上に awk があり，これらのフィルターツールにおいて，効率良く文字列を検索するための基本的かつ一般的な概念といえましょう．そこで，正規表現についてまとめておきます．まず，以下の文字は正規表現において特別な意味を持つメタキャラクタです．

```
\ ^ $ .  [ ] | ( ) * + ?  +
```

これらの文字を文字通りに解釈させるには \ を前に付けなければなりません (リテラル文字)．AWK では，元祖 awk に対して GNU awk の拡張モード，POSIX 互換モードでしか動作しないものもありますから，注意しましょう．

ところで，前節の 2 つの正規表現ですが，もうお分かりですね

```
$1 ~ /^d/     第 1 フィールドが文字 d で始まっている
$NF ~ /\/$/   最後のフィールドが文字 / で終っている
```

という意味になります．

表 2　awk の正規表現一覧

表記	意味
c	メタキャラクタでない文字 *c*
c	メタキャラクタを含めて文字 *c* 自身（リテラル文字 *c*）
.	任意の 1 文字
^	行頭，または文字列の先頭
$	行末，または文字列の終端
[*chras*]	*chars* に含まれる文字のうちどれか 1 文字
[^*chars*]	*chars* に含まれない文字（上の場合の否定ではないことに注意）
[*c1*-*c2*]	*c1* から *c2* までの範囲にある 1 文字
r1 | *r2*	*r1* または *r2*
r1 r2	*r1* の直後に *r2* が続くもの
*r**	*r* の 0 回以上の繰り返し
r+	*r* の 1 回以上の繰り返し
r?	0 か 1 回の *r*
(*r*)	正規表現 *r* のグループ化
gawk で –posix または –re-interval 指定時のみ有効	
r{*n*,*m*}	*r* の *n* 回以上 *m* 回までの繰り返し
r{*n*,}	*r* の *n* 回以上の繰り返し
r{*n*}	*r* の *n* 回の繰り返し

パターンの種類

さて，パターンには正規表現による文字列照合以外に

式，複合，範囲，BEGIN，END，

があります．これらを簡単に説明します．

式： いわゆるプログラミング言語における**式**で，式の集まりが**文**となります．比較演算子

```
> >= == <= < ! ~ !~
```

などを用いて，

```
$ awk '!/root/' awk.data
$ awk '$3 == "root"' awk.data
$ awk '$5 > 0 { print $5,$NF }' awk.data
$ awk 'NR <= 4' awk.data
```

などのように用います．2 番目の例は文字列として等しいことを調べるので，誤って，空白を入れて

```
$ awk '$3 == " root"' awk.data
```

とすると，等値ではありませんから，どの行も表示されません．最後の例は awk.data の最初の行 (NR=1) から 4 行目 (NR=4) までを表示します．すなわち，“head -4 awk.data” と同じ結果を得ます．

複合： 複雑な条件は，式を組み合わせて表現する必要があります．そこで，論理演算子

```
&& ||
```

を用いて，式を複合化します．

```
$ awk '/ro/ && $5 >= 1024' awk.data
$ awk '$NF ~ /^\./ || $1 ~ /^d/' awk.dat
```

範囲： 2 つのパターンで行範囲を指定します．すなわち，パターン 1 が現れた行から，パターン 2 が現れる行までを適合範囲とします．典型的な例は,

```
$ awk 'NR == 2, NR == 6' awk.data
```

というもので，2 行目から 6 行目までを表示することになります．この場合は明らかに適合範囲は 1 つだけですが，複数回適合することもありえます．パターン 1 やパターン 2 に適合する行が多い場合には，どの行からどの行まで適合するか予想しにくいです．例えば,

```
$ awk '/^d/, /^-/' awk.data
```

とすると，たぶん複数の範囲が表示されますが，どうしてそういう結果になったかは相当判りずらいでしょう．

```
$ awk '/^d/, /^-/{ print $1; if (/^-/) \
> print "END" }' awk.data
```

とすれば，判り易くなったと思います．

BEGIN と END： awk を用いて "`head -n`" を実現することは簡単でした．では "`tail -n`" を実現するにはどうしたらよいでしょうか？ 一行ずつ読み込んで処理を実行する awk にとっては難しい問題です．最後の行数は最後まで読み込まないと分からないからです．それではまず最後まで読み込んでと，あれ終了しちゃった... では困るので，特別なパターン **END** があります．同様に，一行ずつ処理を開始する前の初期化を行うために **BEGIN** パターンがあります．

BEGIN は各行に対する処理の前に一度だけ，END は各行に対する処理が終了した後に一度だけアクションを実行させることができます．BEGIN の例としては

```
$ awk 'BEGIN{ FS = ":" } { print "$HOME of "$1\
> " is "$6 }' /etc/passwd
```

などが有名です．フィールドの区切子をコロン ':' に変更して，ユーザーの名前と login 時の `$HOME` を表示しています．END の例は

```
$ awk '$3~/root/{ n++ } END{ print n }' awk.data
```

で理解できるでしょう．文字列 root を含む行数を最後に一回だけ表示させることができます．

プログラムファイル

スクリプトが長くなってくると，コマンドライン上の編集が大変です．そこで，長いスクリプトはファイルに記述しておき,

```
$ awk -f filename.awk
```

のように，引数として与えます．

アクション

パターンの次は当然アクションの説明なのですが，許される式や文の種類と記述法が C 言語に似ています．そこで，網羅的な紹介は省き随時説明を加えようと思います．

表 3 awk の主な組み込み文字列関数

関数	機能
`sub`(r, s, t) `sub`(r, s)	文字列 t を対象にしてその中の正規表現 r に一致する最左最長の部分文字列を，文字列 s に置換する．t が省略された場合は $0 が対象となる．
`gsub`(r, s, t) `gsub`(r, s)	文字列 t を対象にしてその中の正規表現 r に一致する部分文字列を，全て文字列 s に置換する．t が省略された場合は $0 が対象となる．
`index`(s, t)	文字列 s の中に文字列 t が始めて現れる位置を返す．t がない場合は 0 を返す．
`match`(s, r)	文字列 s の中で正規表現 r に適合する位置を返し，`RSTART` と `RLENGTH` を設定する．適合しない場合には 0 を返す．
`split`(s, a, c) `split`(s, a)	文字列 s を文字（列）c を区切子として分解し，配列 a に格納する．c が省略された場合は FS が使われる．
`length`(s)	文字列 s の長さ．
`substr`(s, n, m) `substr`(s, n)	文字列 s の n 番目から始まる長さ m の文字列．m が省略された場合には s の末尾までの文字列．

文字列関数： 表 3 のような，C 言語に比べて高級な文字列関数があります．

文字列関数を使った簡単な例を示します．どの配布系にもある/usr/doc/util-linux-2*/README.hwclock をデータファイルとしますから，(間違って変更しないように) コピーして使いましょう．

```
match($0,/clock/) > 0 {
  printf "%2d 行目%2d 文字目から %s\n",
          NR, RSTART, substr($0,RSTART)
}
```

というの内容で strfunc.awk を作成して，次のように

実行します．なお，**日本語のメッセージを含める場合には必ず拡張 UNIX コードで保存**してください．

```
$ awk -f strfunc.awk README.hwclock
```

clock という文字が現れた位置から行末までが表示されました．ところで，`Clock` や `CLOCK` なども同じ単語として認識させたい場合がありませんか？元祖 awk では無理ですが，gawk では 次のように (GNU 拡張) 組み込み変数 `IGNORECASE` を 0 以外に設定することで可能です．

```
BEGIN { IGONRECASE = 1 }
```

gawk の GNU 拡張を抑止して，元祖 awk の互換モードで使うには

```
$ awk --compat -f strfunc.awk README.hwclock
```

とオプション指定します．

出力のリダイレクトとパイプ： リダイレクト演算子 `>` と `>>` によって，標準出力以外のファイルに出力をすることができます．また，パイプ演算子 `|` によって，出力をパイプに流すこともできます．

```
$ awk '{print $NF > "files";
> print $3 | "sort > users" }' awk.data
```

のように書くことができます．もっとも，なるべく標準出力を使って

```
$ awk '{print $3}' awk.data | sort > users
```

のように実行するのが UNIX 流でしょう．したがって，出力先が複数となる場合 (シェルでは難しい) にスクリプト内部で振り分けるという方針が正しいのかもしれません．

入力： 現在の入力の他に，ファイルやパイプからレコードを読み込むことができます．その場合には `getline` 関数を使います．

`getline <` *file*
`getline` *var* `<` *file*
`"`*command*`" | getline`
`"`*command*`" | getline` *var*

例えば，

```
$ awk '{"date" |getline; print; close("date")}'
```

とすると，Enter キーを押す毎に日時を表示します．一度使った (open) したコマンド名は close で一端閉じないと，再度同じ名前で使うことはできません．

ユーザー定義関数

ユーザー関数は以下の形式で定義できます．

```
function 名前 (引数) {
  文
}
```

とても簡単な例 (`userfunc.awk`) を示します．

```
# userfunc.awk: User defined function.
function prnast (x) {
  z = -1.0
  while (z < x ) {
  printf "%c", " "; z += 1/24.0
  }
  printf "*\n"
}

BEGIN {
  PI = 4*atan2(1,1);
  for (j = 0; j < 24; j++) {
    prnast(sin(j*PI/12));
  }
}
```

ユーザー定義関数 prnast は x の値に応じてアスタリスク記号 '*' をコンソール画面に打ち出します．BEGIN だけで終りですから，入力データは必要ありません．

```
$ awk -f userfunc.awk
```

とすると，図 1のように sin 関数のグラフを表示します．

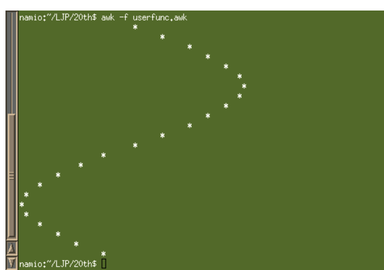

図 1 AWK ユーザー定義関数の例：コンソール画面への sin 関数の打ち出し

活用編

AWK の書籍には，データ処理の実例がたくさん載っていますが，文科系のレポートに関するものが多いようです．それは一般読者層を考えれば当然でしょうが，ここでは筆者の日常からということで理工系の論文を対象にした例を述べます．すなわち，実験データを表や図に自動変換するスクリプトの紹介をします．

TeX の表作成

理工系論文においても，実験データなどを整理してある表はとても有効な表現手段です．しかし，TeX では表のソースの記述は結構面倒です．当然スクリプトで処理することになりますね．TeX の書式はここでは詳しく述べませんが，問題となることが一つあります．TeX と シェルのメタキャラに注意して書き出す必要があるということです．例えば，TeX のコマンドは \ で始まりますが，この記号はシェルのメタキャラですから，\\ とエスケープして書き出さないといけません．リスト 1 のプログラムは，awk.data を変換し，ファイル名，ユーザー名，属性の一覧表の TeX ソースを作成します．

BEGIN の 2 行目 `outf` の名前は文字列を並べて (加える)，新しい文字列にする**連接** (concatenation) を用いています．これは C 言語にはない便利な演算です．

リスト 1　textable.awk

```
BEGIN{
  match(ARGV[1], /\./)
  outf = substr(ARGV[1], 1, RSTART)"textbl"
  print "\\documentclass{jarticle}" >outf
  print "\\begin{document}" >outf
  print "\\begin{table}" > outf
  print "\\begin{tabular}{|l|l|l|}" >outf
  print "\\hline" > outf
  print "ファイル名 & ユーザー名 & 属性\\\\" >outf
  print "\\hline" > outf
}
NR == 2, NR == 30 {
  gsub(/_/,/\_/,$NF) # '_' should be '\_' in TeX
  print $NF "&" $3 "&" $1 "\\\\" >outf
}
END{
  print "\\hline" > outf
  print "\\end{tabular}" >outf
  print "\\end{table}" >outf
  print "\\end{document}" >outf
  close(outf)
}
```

拡張子 .textbl (.tex は危険なので) を付けたファイルを pLaTeX で処理します．

```
$ awk -f textable.awk awk.data
$ platex awk.textbl
$ xdvi awk.dvi
```

すると，次のような表が作成されていることを確認できます．

ファイル名	ユーザー名	属性
./	matuda	drwxrwxrwt
../	root	drwxr-xr-x
.X0-lock	root	-r–r–r–
.X11-unix/	root	dr-xr-xr-t
.font-unix/	root	drwxrwxrwx
.kon1=	matuda	srwxr-xr-x
……		

Gnuplot との連携

パイプを使って，グラフ作成ツール gnuplot に図を描かせてみましょう．wf.data は主だった金属の元素記号，熱伝導率 κ，電気抵抗率 ρ をまとめたものです．熱を良く伝える金属は電気も良く伝えるという法則があって，それを図で示そうというわけです．素直に電気伝導率 σ を与えなかったのは，単に逆数 $\sigma = 1/\rho$ を計算するだけですが，awk や gnuplot でデータの簡単な数値処理が可能であることを示したかったからです．

wf.awk について説明をします．5 行目で名前を "gnuplot -persisit" (実際はプロセスですが) と定義し，6 行目以下でそれにパイプを通じて命令を送っています．実は wf.data のデータ散布図自身は gnuplot だけで可能なのです (wf.awk の 7 行目のように一行で済みます)，がしかし，各点にラベルを付けることが gnuplot ではどうしてもできません．変数への文字列代入ができないからです．そこで，awk の助けを借りて，13〜16 行目のように gnuplot のラベル生成命令 "set label ..." に対して，固定文字列を与えるように組んで成功します．

リスト 2　wf.data

```
# 元素の電気抵抗率 ρ [10^{-8} ohm m] と
# 熱伝導率 κ [J/(m s K)]
# Wiedemann-Franz の法則
#########################################
Ag  1.59  420
Cu  1.69  395
Au  2.44  298
Al  2.66  223
W   5.5   167
Mg  4.46  160
Mo  5.78  147
Ni  7.8   92.4
Fe  10.7  75.6
Co  5.68  69.3
Ta  13.6  54.6
In  8.8   23.9
Ti  42    17.1
```

リスト 3　wf.awk

```
1: BEGIN {
2:   SPC = 0.02
3:   XMAX = 0.7
4:   YMAX = 500
5:   GNUPLOT = "gnuplot -persist"
```

```
 6 :   print "set size square" |GNUPLOT
 7 :   print "plot 'wf.data' using (1.0/$2):3 \
 8 :                        lw 2 pt 21 " |GNUPLOT
 9 :   print "set nokey" |GNUPLOT
10 :   printf "set xrange [0:%f]\n", XMAX |GNUPLOT
11 :   printf "set yrange [0:%f]\n", YMAX |GNUPLOT
12 : }
13 : !/^#/ {
14 :   printf ("set label %d '%s' at first %f,%f \
15 :     left\n", NR, $1, 1.0/$2+SPC, $3) |GNUPLOT
16 : }
17 : END {
18 :   print "replot" |GNUPLOT
19 :   print "set output 'wf_0.obj'" |GNUPLOT
20 :   print "set term tgif 'Helvetica' 24" |GNUPLOT
21 :   print "replot; quit" |GNUPLOT
22 :   close(GNUPLOT)
23 : }
```

wf.awk を以下のように実行すると

```
$ awk -f wf.awk < wf.data
```

“wf_0.obj” という Tgif のソースができますから，Tgif 上で軸名などを書き入れて “wf.obj” に別名保存し，図2のような結果を得ます．

この場合にも，wf.awk 内のパイプコマンドを全て消したスクリプトを wf1.awk (標準出力を使うことになります) として，

```
$ awk -f wf1.awk < wf.data | gnuplot -persist
```

と実行することもできます．むしろこの方が UNIX 的には好ましいかもしれません．

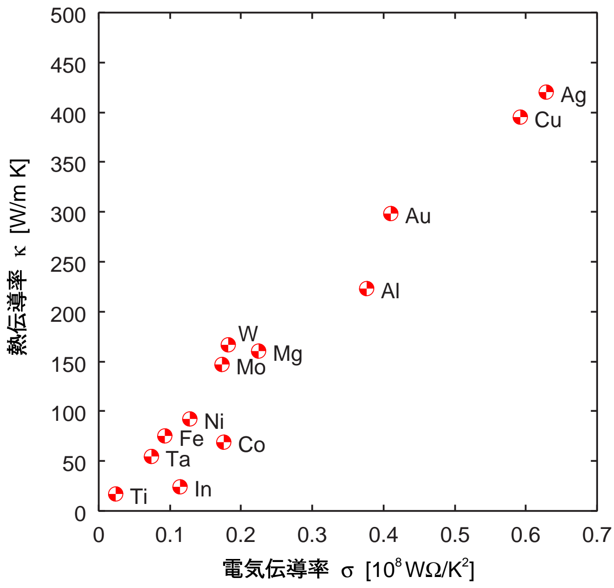

図 2 AWK+Gnuplot で各データ点へのラベル付けを行った例

邪道編

元祖 awk を B.W. Kernighan 氏は今もなお保守しています [6][W3]．それらのソースは読みやすく，組み込み関数を簡単に追加することができます．組み込み数学関数については GNU でも拡張されておらず，表4のままです．これらの関数では表すことのできないベッセル関数 (j0(x),j1(x),y0(x),y1(x) として libm には含まれています) を使えるようにしてみましょう．もちろん，これは AWK の仕様を越えますから，本当に自分の楽しみのためだけの私的な改造です．

表 4 awk の主な数学関数

関数	意味
int(x)	小数点以下を切捨てた整数値
atan2(y, x)	逆正接関数 $\arctan(y/x)$
cos(x)	余弦関数
sin(x)	正弦関数
exp(x)	指数関数
log(x)	自然対数関数
sqrt(x)	平方根
rand()	[0,1] 区間の疑似乱数発生
srand(i)	i を種にした rand() の初期化

手に入れたソースを展開し，以下のように awk.h lex.c run.c に記述を追加することで好みの関数を登録できます．

1. awk.h の 112 行目以下に 14 個の組み込み関数が定義されていますから，次のように追加します．
```
#define FJ0 15
#define FJ1 16
```
2. lex.c のキーワード定義配列に以下の要素を追加します．
```
{"j0", FJ0, BLTIN},
{"j1", FJ1, BLTIN},
```
3. 最後に run.c の 1650 行目から始まる bltin 関数の case 文に追加します．
```
case FJ0:
  u = j0 (getfval (x));
  break;
case FJ1:
  u = j1 (getfval (x));
  break;
```

あとは，make 一発，あっという間にコンパイル終了です．できあがった実行バイナリ a.out を名前を “noawk” としてインストールしましょうか．

```
$ install -c -s ./a.out /usr/local/bin/noawk
```

ベッセル関数 j0 が組み込まれたことを確かめた実行例を示します．

```
$ noawk 'BEGIN{ while (x<=1.0) {printf"%f, %f\n",
> j0(x), j1(x); x+=0.2} }'
1.000000, 0.000000
0.990025, 0.099501
0.960398, 0.196027
0.912005, 0.286701
0.846287, 0.368842
0.765198, 0.440051
```

高級プログラマブル関数電卓として使えますね．

参考文献

[1] FAQ 集：http://www.faqs.org/faqs/computer-lang/awk/faq/ W3

[2] 自習コース： http://cit.rcc.on.ca/cscourse.htm W3

[3] A.V. Aho, P.J. Weinberg, B.W.Kernighan 著 足立高徳 訳,『プログラミング言語 AWK』(トッパン)

[4] D. Dougherty 著 福崎俊博 訳,『sed & awk プログラミング』(アスキー出版局) W3

[5] A.D. Robbins,『GAWK: Effective AWK Programming』(Free Software Foundation)：http://www.gnu.org/manual/gawk/index.html W3

[6] 元祖 awk のホームページ：http://cm.bell-labs.com/cm/cs/awkbook/index.html W3