2016 年 07 月 08 日

計算機実習 III (2016 年度)
第 11 回: AWK その 5

(http://takeno.iee.niit.ac.jp/%7Eshige/
math/lecture/comp4/comp4.html)

目 次

1 正規表現 その 2

前回は、表 1 の最初の 7 つのメタ文字を紹介したので、今回は残りの 4 つのメタ文字について説明する。

メタ文字	意味
.	任意の 1 文字
?	この直前の文字が 0 個または 1 個ある
*	この直前の文字が 0 個以上続く
+	この直前の文字が 1 個以上続く
¥	エスケープ文字
^	文字列の先頭 (文字は意味しない)
$	文字列の末尾 (文字は意味しない)
[ ]	[ ] 内の文字のいずれか 1 文字
[^]	[^] 内の文字以外のいずれか 1 文字
( )	( ) 内の正規表現をグループ化
|	OR (主に ( ) 内で用いる)

表 1: メタ文字一覧

[ ] は、その中に書き並べた複数の文字のうち「いずれかの 1 文字」を表し、1 文字毎の OR を意味することになる。文字を書き並べる順序はほぼ何でもよいが、- は後述するように特別な意味で使われるので、文字の候補として - を入れる場合は、それを先頭に書く。

[^] は [ ] の逆を意味し、すなわちその中に書き並べた複数の文字には含まれない文字 1 文字にマッチする。[ ], [^] の中には、指定する文字を書き並べる以外にも、- で (ASCII コード順に) 範囲指定することもできる。

正規表現の例	意味
[a1c2A]	a, c, 1, 2, A のいずれか 1 文字
[a-c1-3A]	a, b, c, 1, 2, 3, A のいずれか 1 文字
[^a-c1-3A]	a, b, c, 1, 2, 3, A 以外のいずれか 1 文字
[0-9A-Za-z]	アルファベットか数字のいずれか 1 文字
[-axf]d	"-d", "ad", "xd", "fd" のいずれか
[01][A-C]	"0A", "0B", "0C", "1A", "1B", "1C" のいずれか
[^sf]printf	"sprintf", "fprintf" 以外の "printf"

[ ] は複数の 1 文字の OR を意味するが、1 文字ではない文字列の OR (いずれか) を表現するには、| を使用する。これは、始まりと終りがわかるよう、通常 ( ) 内で使用する。なお、( ) は、特に特定の文字を意味はせず、正規表現をグループ化する (ひとかたまりとする) のに使用する。

正規表現の例	意味
(ABC|12|pq)	"ABC", "12", "pq" のいずれか
[aA]¥.(ttf|TTF)	"a.ttf","A.ttf","a.TTF","A.TTF" のいずれか
(9|10)¥/(3|10)	"9/3", "9/10", "10/3", "10/10" のいずれか

[ ], [^] で囲んだ正規表現は全体で 1 文字を意味するので、この後ろにも前回の ?, *, + をつけて、その繰り返しを意味させることができる。また、( ) で囲んだグループ化された正規表現にも、後ろに ?, *, + をつけて、そのグループ化された正規表現の繰り返しを意味させることができる。

正規表現の例	意味
[01]+	0 と 1 だけからなる 1 文字以上の任意の文字列
(01)+	"01" の繰り返し ("01", "0101", "010101" 等)
(01|10)+	"10" か "01" の繰り返し ("0110", "011010" 等)

以下に、いくつか注意やヒントを上げる。

- [ ] と (|) は、いずれも OR であるが、例えば正規表現

```
[34][56][78]
```

は、357,358,367,368,457,458,467,468 の 8 通りのいずれかの文字列を意味し、以下のいずれも一応それと同じ意味になる。

```
(357|358|367|368|457|458|467|468)
(35|36|45|46)[78]
[34](57|58|67|68)
```

しかし、もちろん最初に示したものが一番無駄がなくてよい。

- 『akb48, AKB48, ngt48, NGT48 のいずれかが含まれる文字列』を意味する正規表現の条件式は、

```
/(akb48|AKB48|ngt48|NGT48)/
```

でもよいが、違っているのは "48" の前の 3 文字だけなので、

```
/(akb|AKB|ngt|NGT)48/
```

の方がよい。なお、アルファベットの部分がいずれも大文字も小文字もありうる場合は、

```
/([aA][kK][bB]|[nN][gG][tT])48/
```

と書けば「AKb48」や「NgT48」などの文字列にもマッチする。

- [ ], (|) の後ろに *, + をつけて繰り返しを表した場合は「同じ文字」が繰り返される必要はなく、「OR という状態」の繰り返しを意味する。例えば、

```
/[25]+/
```

は、"22", "555", "2222" のような文字列だけではなく、"25", "55225" のような文字列にもマッチする。

- 『2 桁の正の 10 進整数 (10 ~ 99) を含む文字列にマッチする正規表現』の条件式は、1 文字目は 1 ~ 9 のどれでもよく、2 文字目は 0 ~ 9 のどれでもいいので、

```
/[1-9][0-9]/
```

でよい。一方、『2 桁「以下」の正の 10 進整数 (1 ~ 99) の文字列に「のみ」マッチする正規表現』の条件式は、1 文字目は 1 ~ 9 だが、2 文字目は「ない」か 0 ~ 9 のいずれか、となるので、? を用いて

```
/^[1-9][0-9]?$/
```

となる。なお、こちらは「のみ」という条件もついているので、先頭に文字列の先頭を意味する「^」、最後に文字列の終わりを意味する「$」が必要になる。

さらに、これを『2 桁以下の「0 以上」の 10 進整数 (0 ~ 99) の文字列にのみマッチする正規表現』の条件式に変えると、また少し面倒になる。

単純に 1 文字目の 1 を 0 に変えて「/^[0-9][0-9]?$/」としてしまうと、「"03"」のような望ましくない文字列にもマッチしてしまう。すなわち、1 文字目が 0 の場合には 2 文字目はあってはならず、その場合だけ他とは違う「例外的な扱い」になる。よって、この場合の正規表現は、(|) も用いて次のようにする必要がある。

```
/^(0|[1-9][0-9]?)$/
```

これは、「/^0$/」と「/^[1-9][0-9]?$/」の OR を意味する。

2 正規表現を使う関数

AWK には、match() 以外にも、正規表現を使う表 2 のような関数がある。

関数	意味
sub(r, s1 {, s2})	(文字列 s2 内の) 正規表現 r に最初にマッチする部分を s1 に置換する
gsub(r, s1 {, s2})	(文字列 s2 内の) 正規表現 r にマッチするすべての部分を s1 に置換する
split(s, h {, r})	第 3 引数に正規表現 r を指定するとそれを区切り文字として配列に切り分ける

表 2: 正規表現を使う関数

sub(), gsub() は、第 3 引数 s2 を省略した場合は文字列 $0 の置換を行う。例えば、スクリプト

```
{ gsub(/。/, ". "); gsub(/、/, ", "); print }
```

は、データファイルの全角句読点「。」「、」を「.」「,」に変換して出力する。なお、これまでの関数とは違い、sub(), gsub() は s2 (省略した場合は $0) を「直接書き換える」ので、元の文字列が必要ならば、これらの関数にかける前にそれを別の変数に保存しておく必要がある。

前回、「#」が行頭にある行を削除する 5 文字のスクリプトを示したが、さらに行の途中にある「#」から行末までもカットするなら、

```
!/^#/ { sub(/#.*$/, ""); print }
```

とすればよい。「/#.*$/」は「#」から行末までを意味し、それを空文字列「""」に置換するので、結果としてその部分を削除することになる。マッチしなければ何もしない。sub(), gsub() は、このように部分文字列の削除にも使える。

第 8 回で紹介した split() は、文字列 (第 1 引数) を配列 (第 2 引数) に切り分ける関数で、第 3 引数に何も指定しなければ空白とタブを単語の区切りと見て切り分けるが、その区切りを第 3 引数に正規表現を指定することで変更できる。例えば、

```
BEGIN { N = split("2014-06-17:10:40:00", h, /[-:]/)
    for (j=1; j<=N; j++) print h[j] }
```

とすると、第 3 引数の /[-:]/ により - または : が単語の区切りとなり、対象文字列の "2014-06-17:10:40:00" の年、月、日、時、分、秒がすべてバラバラになって、h[j] には

```
h[1]="2014", h[2]="06", h[3]="17", h[4]="10", h[5]="40",
h[6]="00"
```

のように保存される。

3 処理の中断

各行の処理の途中でそれを中断する命令に next, exit がある。(表 3)。

命令	意味	その次の作業
next	現在の入力行の処理を中断	スクリプトの先頭に戻り、次の入力行の処理を行う
exit	現在の入力行の処理を中断	データ処理を終了して、END ブロックに進む (あれば)

表 3: next と exit

next は C 言語にはなく、第 9 回の資料の図 1 でいうと、(4) を中断して (2) へ進むもので、if 文の else を減らしたり、多段の if, for などから一気に抜けるのに利用できる。

例えば、一つ前の行との差の値を出力するスクリプト

```
{ if (NR == 1) prev = $1
    else { print $1 - prev; prev = $1 } }
```

は、next を使って (else を使わずに)、

```
{ if (NR == 1) { prev = $1; next }
    print $1 - prev; prev = $1 }
```

と書くことができる。

次のスクリプトは、1 列目の最大値と最小値を表示する。

```
{ if (NR == 1) { max = $1; min = $1; next }
    if ($1 > max) { max = $1 ; next }
    if ($1 < min) min = $1 }
END { print max, min }
```

これも、if が真の場合は通常ブロック内の下の処理が必要ないので、next で次の行の処理にジャンプすることができる。

一方、exit は、第 9 回の資料の図 1 の (4) からデータの残りの行は無視して一気に (5) へ進むもので、C 言語と同様、主にエラー処理などで利用する。ただし、C 言語の exit() とは違い、完全にそこで終了するわけではないことに注意が必要。

- exit を実行したのが END ブロック内でない場合は、もし END ブロックがあれば END ブロックへ進み、END ブロックがなければそこで終了する。
- exit を実行したのが END ブロック内ならば、そこで終了する。

よって、エラー処理の目的で exit を使う場合、END ブロックでは exit で飛んできたのか、それとも行の読み込みが終わって来たのかをわかるようにするために、途中で END ブロックに飛ぶ場合は目印を設定する必要がある。例:

```
{   if (NF < 2) { iserror = 1; exit }
    x += $1; y += $2 }
END { if (iserror) { print "列不足", NF, NR; exit }
    printf "平均値:  %f, %f¥n", x/NR, y/NR }
```

このスクリプトでは iserror という変数を使って、exit によって END ブロックに飛んできたのか (iserror=1)、行の読み込みが全部終わってここに来たのか (iserror=0) を判別している。iserror は初期化していないので初期値は 0 で、exit で END ブロックにきたのでなければ iserror は偽 (0) となり、END ブロックの if 文には入らない。

END ブロックの if 文内では、エラーメッセージと exit による中断時点での NF, NR の値を出力し、ここで再び exit を行っているが、この exit は END ブロック内なのでそこで完全に終了する。

一方、exit でなく END ブロックに来た場合は、NR はデータの全体の行数を意味するので正しく平均値が求まる。

ループ内での処理を中断する continue, break は、AWK でも C 言語と同様に使用できるが、next と exit は、それぞれ「行読み込み」というループ (第 9 回資料図 1 の (2) から (4) のループ) に関する continue と break に対応する、と見ることもできる (表 4)。

	for/while ループ	行読み込みループ
現在のループ処理のみ中断	continue	next
ループ全体を中断して外へ	break	exit

表 4: ループの中断命令群

コラム: フリーソフトの文化

「フリーソフト」という言葉は多くの人が知っているだろうが、実は「フリーソフト」にはいくつかの意味がある。例えばよく見られるものに以下のようなものがある。

1. 無料でダウンロードして無料で使える (free = 無料)
2. 試用は無料だが、本格的な利用は課金される (free = 試用の自由)
3. ソースを見る、改変する、再配布することも自由 (free = 改変の自由)

MS–Windows 上の「フリーソフト」は 1. か 2. の意味のものが多く、個人で作成しているフリーソフトでもソースを見せない形式、実行バイナリだけの配付のものが多い。

フリーソフトの文化は、MS–Windows よりむしろ Unix という OS 上で発展したもので、現在でも Unix では多くのソフトが 3. の形、つまり C 言語のソースファイルの形で配付されている。そのようなフリーソフトを、以前は「パブリックドメインソフトウェア (PDS)」と呼んだ時期もあったが、これは著作権を放棄するという意味合いが含まれるため、現在では「オープンソースソフトウェア (OSS)」と呼ぶことが多い。

OSS のソフトは、個人や趣味で集まった数名が作成したものも多いが、現在は会社や団体で作成・管理しているものもあるし、インターネット上の OSS 開発プロジェクト

の支援サイト (SourceForge.net, GitHub 等) 上で多くのボランティアの開発者によって開発されるものもあり、現在でも非常に多くの OSS がネットワークを通じて世界中の人々により開発、更新され、利用されている。

OSS のメリットには、以下のようなものがある。

- ユーザも開発に参加できるのでユーザの声を反映したものになりやすい
- ソースが公開されているので不具合の改変も随時行われる
- ソースを見て勉強することができる
- 開発に参加しなくても自分の使いやすいように改変できる
  (ただし、流用したり、改変を公開する場合はライセンスに従う必要がある)

以前の PDS とは違い、現在の OSS にはしっかりとした著作権 (ライセンス) が明確に決められていて、良く用いられるライセンスには数種類があるが、その中でも有名なのが、GNU というプロジェクトが提唱する「GPL (GNU Public Licence)」というライセンスである。GPL は「コピーレフト」という言葉で説明される、自由に対する要求がとても強いライセンスであり、逆にそのために GPL とは別なライセンスを選択する OSS もあるくらいであるが、TV などの電化製品のマニュアルにそれらが内部で使用しているソフトの GPL 表記が書かれていたりする位、現在は色々な場面で GPL に基づくソフトウェアが利用されている。

「GNU」とは、世界的に有名なハッカー (注:「ハッカー」という用語は元々悪人を指す言葉ではない) であるリチャード・ストールマンが設立した FSF (Free Software Foundation) が掲げるプロジェクトで、黎明期の途中から閉鎖的になっていった Unix (のライセンスを持っていた AT&T) に対抗して、OS も含めコンピュータに必要なソフトウェアをすべて OSS として作成することを目標としたものであり、現在も GNU プロジェクトは OS を含む非常に多くの重要なソフトウェアを公開し続けている。

gawk (= GNU awk) もその名前からわかる通り、GNU 版の AWK ソフトウェアで、GPL に従う OSS である。GNU 版の awk は、オリジナルの A, W, K らによる AWK を色々な面で拡張したものになっていて、「AWK」としては世界中でほぼ標準的に GNU awk が広く使用されている。