情報通信コース実験 II

スクリプトプログラミング (担当 竹野、2020 年度)

(http://takeno.iee.niit.ac.jp/%7Eshige/
math/lecture/jikken1/jikken1.html)

第 3 回 AWK とフィルタ処理

1 AWK に用意されている主な関数

AWK に用意されている主な関数を表 1 に示す。

三角関数の角の単位はラジアンなので、x の単位が度である場合は、

```
BEGIN { x=15; pi=3.1415926535; print sin(x*pi/180) }
```

のように $\pi/180$ 倍して変換する必要がある。なお、円周率 π は AWK では定義されていないので、上のように実際の値として定義するか、または atan2() を利用して「pi=atan2(0,-1)」とする。

AWK に用意されている数学関数は C 言語に比べれば少ないが、以下の公式を用いればいくつかの関数は補うことができる。

$$\begin{aligned}
\tan x &= \frac{\sin x}{\cos x} \quad (\cos x \neq 0 \text{ のとき}), & \sin^{-1} x &= \mathrm{atan2}(x, \sqrt{1-x^2}) \\
\log_a x &= \frac{\log x}{\log a} \quad (a \neq 1,\ a > 0 \text{ のとき}), & \cos^{-1} x &= \mathrm{atan2}(\sqrt{1-x^2}, x) \\
\sqrt[n]{x} &= x^{1/n} \quad (x > 0,\ n \geq 2), & \tan^{-1} x &= \mathrm{atan2}(x, 1)
\end{aligned}$$

例:

```
BEGIN { pi = atan2(0, -1)
    print "簡易三角関数表"; print "度 cos   sin   tan"
    for (j=0; j<90; j++) {
        printf "%2d", j
        t = j*pi/180; x = cos(t); y = sin(t); z = y/x
        printf " %5.3f %5.3f %5.3f¥n", x, y, z }
    printf "%2d %5.3f %5.3f %5s¥n", 90, 0.0, 1.0, "--" }
```

数学関数	
関数	返り値等の意味
`sin(x)`, `cos(x)`	x のサイン, コサインの値 (x はラジアン)
`atan2(y,x)`	$\tan\theta = y/x$ となる θ の値 ($-\pi \leq \theta \leq \pi$)
`exp(x)`, `log(x)`	e^x ($e = 2.71828\ldots$), $\log_e x$ ($= x$ の自然対数)
`sqrt(x)`	$\sqrt{x}$
`int(x)`	x の整数部分 (0 に近い方) を返す
乱数関数	
`rand()`	0.0 以上 1.0 未満の一様疑似乱数を返す
`srand()`	`rand()` の初期値を設定 (返値なし)
時刻関数	
`systime()`	1970 年 1 月 1 日 00:00:00 から現在までの秒数
文字列処理関数	
`length(s)`	文字列 s の長さを返す
`tolower(s)`	s のアルファベットを小文字にした文字列を返す
`toupper(s)`	s のアルファベットを大文字にした文字列を返す
`sprintf(f, ...)`	`printf()` の出力と同じ文字列を返す
`index(s1, s2)`	文字列 $s1$ 内の $s2$ の先頭位置を返す
`substr(s, n {,m})`	文字列 s の場所 n から長さ m (デフォルトは最後まで) の部分文字列を返す
`split(s, a)`	文字列 s を半角空白区切りで切り分けて配列 a に保存し、切り分けた個数を返す

表 1: AWK が提供する主な関数

注意:

- C 言語では `1/3` と書くと 0 となる (整数の割り算) が、AWK はすべて実数計算なので 0.333... となる。整数の割り算値を得るには `int(x/y)` とする。
- x が整数 (正確には小数部分が 0 の実数) かどうかは、「`int(x) == x`」という条件式で判別できる。
- 正の実数 x を四捨五入して整数 n にするには、`n=int(x+0.5)` とすればよい。

`rand()` が生成する乱数は 0 以上 1 未満の実数なので、整数の乱数を作りたい場合は、`int()` などを利用して変換する。例:

```
BEGIN { srand(); print "4 回ジャンケン"
    s[0] = "グー"; s[1] = "チョキ"; s[2] = "パー"
    for (j=1; j<=4; j++) {
        x = int(rand()*3)
        # 1/3 ずつの確率で x は 0,1,2
        print s[x] } }
```

rand() には引数はなく、呼び出す度に違う値 (疑似乱数列) が返る。ただし rand() の実行前に、その乱数列の初期値を決める srand() を実行しておかないと毎回同じ乱数列が発生してしまう。逆に srand() は rand() を使う度に実行するのではなく、プログラムの中で一回だけ実行する。

「rand()*3」により、0 以上 3 未満の実数乱数が得られるが、int() でその整数部分を取ることで、ほぼ均等に 0, 1, 2 が現れる整数乱数となる。

なお、以下のように if 文で場合分けすれば均等でない乱数にすることもできる。

```
BEGIN { srand(); print "4 回ジャンケン";
    for (j=1; j<=4; j++) {
        x = rand()
        if (x < 0.2) print "グー"; # 確率 0.2
        else if (x < 0.7) print "チョキ"; # 確率 0.5
        else print "パー"; # 確率 0.3
    } }
```

systime() は 1970 年 1 月 1 日からの絶対秒数を返すので、2 つの時刻の差を取ることに使える。例:

```
BEGIN { t = systime() # 最初の時刻
    print "5 秒後に止まります。"
    while (systime() < t + 5) ;
    print "5 秒経ちました。" }
```

while 文の最後に ; がついているのは、この while 文の実行部分がないことを示す (そのため省略できない)。systime() の実行にかかる時間は非常に短く、この while 文のループは、実際には相当な回数実行され、よってかなりの精度で 5 秒後に止まることになる。

文字列を処理する関数も表 1 のように用意されているが、この他にも文字列の連結は単に並べるだけでよい。例えば「s = "abc" "d" "e"」とすると s は"abcde" になり、

「s = "ab" toupper("cd")」とすると s は "abCD" となる。

substr() は文字列から一部分を切り出すのに使用する。例:

```
BEGIN { s = "abcdefghijklmnopqrstuvwxyz"; N = length(s)
    for (j=1; j<=N; j++) {
        c = substr(s, j, 1)
        printf "[%d] %s %s¥n", j, c, toupper(c) }}
```

これは、アルファベットの小文字と大文字を順に表示する。substr(s, j, 1) は、s の j 番目の長さ 1 の文字列、すなわち 1 文字を取り出す。

split() はスペース区切りの文字列を配列に切り分けるので、配列を初期化するのに利用できる。例:

```
BEGIN { sj = "睦月 如月 弥生 卯月 皐月 水無月 文月 葉月"
    sj = sj " 長月 神無月 霜月 師走"
    split(sj, mj)
    se = "January February March April May June July"
    se = se " August September October November December"
    N = split(se, me)
    for (j=1; j<=N; j++)
        printf "%2d: %6s %s¥n", j, mj[j], me[j] }
```

2 行目、5 行目は、それぞれ 1 行目、4 行目で書けなかった残りの部分を連結していて長い文字列にしている。そして、その日本語名と英語名の月名を split() で mj[], me[] に配列化している。split() では配列の添字は、1 番から順番に使われ、例えば mj[1] は「睦月」、me[2] は「February」となる。「mj[1]="睦月"」のように順に 12 個代入するより split() を使う方が楽に初期化できる。

sprintf() は、書式化した文字列を文字列として利用できる。例えば、長さ 20 の空白文字列を作るには「s = sprintf("%20s", "")」でよい。なお printf とは違い、sprintf() には ( ) が必要。

index() は簡単な部分文字列の検索に使えるが、AWK は「正規表現」をサポートしていて、本来は文字列の検索には正規表現や match() を使う方が便利である (が、本実習では説明は省略する)。

2 AWK のフィルタとしての動作

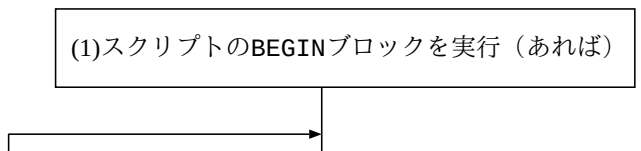
今回より、AWK の行単位のフィルタとしての動作について説明する。この場合、今までのようなスクリプトファイル指定の後ろに、AWK に処理させるデータファイルを指定する:

```
> gawk -f [スクリプトファイル] [データファイル]
```

このように実行すると、AWK は図 1 に示すフローチャートのように、データファイルのデータを 1 行ずつ読み込んで、そのそれぞれの行に対してスクリプトで書いた処理を順番に行う。このように、入力テキストデータを行単位で処理し、結果を画面 (標準出力) に出力するプログラムを **行フィルタ** と呼ぶ。

図 1: フィルタとしての AWK の処理の流れ

AWK スクリプトの構造は、一般には「BEGIN ブロック」、「通常ブロック」、「END ブロック」の 3 種類の実行ブロックからなる。



```
BEGIN {
    [BEGIN ブロックの実行内容]
}
[条件] {
    [通常ブロックの実行内容]
}
END {
    [END ブロックの実行内容]
}
```

注意:

- BEGIN ブロック、通常ブロック、END ブロックはそれぞれ省略可能。
- BEGIN ブロックはデータを読み込み前に最初に 1 度だけ実行される (図 1 の (1))。
- 通常ブロックは、データの行数分だけ繰り返し実行される (図 1 の (4))。
- END ブロックはデータの最後の行に対する通常ブロック処理が終わった後に 1 度だけ実行される (図 1 の (5))。
- 通常ブロックの前に [条件] をつけると (省略可)、その [条件] を満たす行データに対してだけその通常ブロックを実行する。
- [条件] のみで通常ブロックを省略した場合、それは「[条件] { print $0 }」と同じとみなされ、すなわち、[条件] を満たすデータ行がそのまま出力される。

まず図 1 のフローチャートの (3) の部分を説明する。

AWK は、データファイルを 1 行読みこむと、それを空白やタブで区切られた **フィールド** (あるいは **列** とも言う) と呼ばれる単位に分割し、その結果を表 2 のような変数にそれぞれ保存する (バッチファイルのオプションに似ている)。

変数	意味
$[n]	その行の [n] 番目のフィールド文字列 (先頭が 1)
$0	その行全体の文字列
NF	その行のフィールド数
NR	その行の行番号 (先頭行が 1)

表 2: 行の読み込みで設定されるフィールド変数等

例えば、データファイル data1.txt が

```
2016 年 6 月 22 日（水）
2016 年 6 月 23 日（木）
2016 年 6 月 24 日（金）
```

のように 3 行のデータで、各行にはスペース区切りで 7 列のフィールドデータが書かれている場合、例えばこの 2 行目が読み込まれた時点 ((3) の処理のあと) での各フィールド変数の値は、

```
NF=7, NR=2, $1="2016", $2="年", $3="6", $4="月", $5="23",
$6="日", $7="(木)", $0="2016 年 6 月 23 日 (木)"
```

となる (NF と NR は数値、$ 変数は基本的に文字列値)。よって、例えば test1.awk が

```
BEGIN { print "日付は" }
{ print $3, $4, $5, $6 }
END { print "です。" }
```

というスクリプトであるとき、

```
> gawk -f test1.awk data1.txt
```

とすると、

```
日付は
6 月 22 日
6 月 23 日
6 月 24 日
です。
```

と表示される。

以下のスクリプトは、データの 1 列目と 2 列目を数値として足した結果を x に保存しそれを出力する、という作業をすべての行に対して行う。

```
{ x = $1 + $2; print x }
```

なお、print の後ろには数式を書くこともできるので、これは

```
{ print $1 + $2 }
```

というスクリプトでも同じことになる。

```
{ printf "%4d: %s¥n", NR, $0 }
```

というスクリプトは、入力データファイルの行番号を 4 桁で表示し、その後ろに「:」とスペースを 1 つつけて、その後ろに行の内容をそのまま表示する。これにより、元のデータファイルの左に行番号が付加されたものが表示される。この結果を

```
> gawk -f test2.awk hoge1.c > hoge1-c.txt
```

のように、出力リダイレクションを用いてファイルとして保存することももちろん可能である。

$ 変数を使う場合、$ の後ろには 1, 2, 3 などの定数以外に、「j=3; s=$j」のように値が整数である変数や、「s=$(2*j+3)」のように結果が整数となる数式を指定することもできる (ただし数式をかっこで囲む必要がある)。例えば、$NF はその行の最終フィールド文字列、$(NF-1) はその行の最後から一つ手前のフィールド文字列になる。

```
{ for (j=1; j<=NF; j++) print $j }
```

というスクリプトは、データファイルのすべてのフィールド (単語) を、1 行に 1 つずつ出力する。for 文で j を 1 から NF まで変化させることで、$j はその行の最初のフィールド文字列から最後のフィールド文字列の値を順に取ることになるし、行によってフィールド数が異なるデータでも行毎に NF の値は変化するので、正しくすべてのフィールドデータを見てくれることになる。

3 AWK の一行スクリプト

gawk の実行形式として、AWK スクリプトの内容が 1 行で書ける場合 (いわゆる **1 行スクリプト**) は、スクリプトファイルを作らなくても、以下のように実行できる。

```
> gawk "[スクリプトの内容]" [データファイル]
```

すなわち、「-f [スクリプトファイル]」の代わりに、コマンドプロンプト上でスクリプトの内容を " " で囲んで指定する。例:

```
> gawk "{print $2,$1}" data1.txt > data2.txt
```

は、data1.txt のすべての行の 1 列目と 2 列目を入れかえたものを、出力リダイレクト (>) を利用して data2.txt に保存する (出力する)。

この実行形式は、バッチファイル内での AWK の活用の際に良く用いられる。

注意:

- 一行スクリプトの内部で文字 " を使うときは、スクリプト全体を囲む " と区別するために¥" と書く必要がある。
- 条件部分だけのスクリプトでも " " で囲まなければいけない。
- 一行スクリプトの gawk のコマンドを「バッチファイル」内に書く場合は、% を %% と書く必要がある (コマンドプロンプトで実行するだけなら % ひとつ)。

以下に 1 行スクリプトの例をいくつか紹介する (gawk コマンドとデータファイル部分を除いた 1 行スクリプト部分だけ)。

- ```
  "NR<=10"
  ```

  データの最初の 10 行を出力 (条件部分だけのスクリプト)
  なお、これは条件部分だけのスクリプトで、実行部分をつけた以下のものと同等。

  ```
  "NR<=10 {print $0}"
  ```

  ```
  "{if (NR<=10) print $0}"
  ```

- ```
  "NF>0"
  ```

  空行を削除 (空行以外を出力することになるので)

- ```
  "{n+=$1}END{print n}"
  ```

  1 列目の数値の合計を出力 (END ブロックを通常ブロックの後ろに並べて書ける)

- ```
  "{n+=NF}END{print n}"
  ```

  全体の単語数を最後に表示

- ```
  "END{print NR}"
  ```

  全体の行数を表示

- ```
  "END{print $0}"
  ```

  最後の行を表示 ($0 は END ブロックで更新されない)

- ```
  "{print tolower($0)}"
  ```

  大文字アルファベットをすべて小文字に変換
  ```

- ```
  "BEGIN{for(j=1;j<=30;j++)printf ¥"2018-11-%02d¥n¥",j}"
  ```

  11 月の日付一覧を出力 (BEGIN ブロックのみなので入力データは不要)

- ```
  "BEGIN{srand();for(j=1;j<=100;j++) print 5*rand()}"
  ```

  0.0 以上 5.0 未満の実数乱数を 100 個表示

- ```
  "BEGIN{t=systime(); while(systime()<t+10);}"
  ```

  10 秒間何もせずに待つのみ

4 最後に

コマンドプロンプトやバッチファイルは、コマンドをキーボードで入力したり、コマンド名を文字で指定するような方式でコマンドを実行するから、GUI 全盛の現在にはやや面倒くさい、古くさいと感じるかもしれない。

しかし、バッチファイルの簡単な構文や、今回紹介しなかった入力リダイレクションやパイプ機能などを使用することで、小さな単機能のコマンドを組み合わせて使用することができる。よって、簡単な C 言語の知識を有効に生かすには、本来はコマンドプロンプトでバッチファイルを利用するのが一番だと思う。

また、システム管理の現場では、現在でも GUI のツールではなく、コマンドプロンプトを使って作業を行うことが多いそうである。

AWK も、今回紹介した機能以外にも、連想配列やユーザ定義関数、正規表現とそれに付随する関数など、まだまだ多くの有用な機能、スクリプト言語らしい機能を持っている。特に、正規表現や、それによる置換関数 (sub(), gsub()) を用いると、文字列の検索や、文字列の置換、修正などが AWK の一行スクリプトで容易に行え、バッチファイルでのパイプ処理に必要なコマンドがかなり AWK で補える。

なお、「正規表現」は、多くのスクリプト言語で標準的に、そして頻繁に使われているので、スクリプト言語に興味がある人は勉強することを勧める。そして、Perl, Python, Ruby, PHP などの汎用スクリプト言語の学習につなげていってもらいたいと思う。
  ```