2012 年 06 月 15 日

計算機実習 IV (2012 年度)
第 9 回: AWK その 4

目 次

1 連想配列

AWK の配列の添字は、前に紹介したように 0 から始める必要はないが、さらに負の数も、文字列でさえも添字に使うことができる。この性質により、AWK の配列は **連想配列** と呼ばれる。例:

```
week["Sun"] = "日"; week["Mon"] = "月"; ...
week["Sat"] = "土";
```

これを使えば、例えば各行に学生のデータが入っているデータがあり、1 列目が学籍番号、2 列目が性別 (男か女)、3 列目と 4 列目には身長と体重が書かれているとき、AWK で

```
{ N[$2] ++; h[$2] += $3; w[$3] += $4 }
```

のようにすれば、`N["男"]` には男性の人数、`h["男"]` には男性の身長の合計、`w["男"]` には男性の体重の合計がそれぞれ保存される。`h["女"]` 等も同様であり、END ブロックで容易に性別毎の平均等の計算ができることになる。連想配列はこのように項目毎に集計するのにも便利である。

また、for 文には、通常の C と同じ形式の

```
for ([初期設定]; [条件]; [増分設定])
```

以外に、主に連想配列用に

```
for ([変数] in [配列名])
```

という形式があり、[変数] を [配列名] の配列の添字全部を動かして実行することができるようになっている。例えば、

```
BEGIN {
    a[10]="A"; a[13]="D"; a[15]="F";
    for (j in a)
        printf "a[%s] = %s¥n", j, a[j];
}
```

とすると j は 10, 13, 15 を動き、

```
a[13] = D
a[15] = F
a[10] = A
```

のように表示される[1]。

これを利用すると、簡単に逆引きの配列を作ることができる。例えば、上の配列 a[] に対して

```
for (j in a) b[a[j]] = j;
```

とすると、

```
b["A"]=10, b["D"]=13, b["F"]=15
```

という配列 b[] が作られることになる。

また、delete という命令で不要な配列の要素を削除することができる。

- delete [配列要素] : 指定した配列要素を削除する

例えば、上の配列 a[] に対して、

```
delete a[3];
```

とすれば、a[] は要素が 2 つの配列となる。

課題 9-1. a[0]="○", a[1]="一",... a[9]="九" の 10 個の漢数字の配列を

[1] この例のように、for 文のこの形式では、j に代入される添字の順番は、必ずしもソート順にはならない。

作り、for 文を使ってこの逆引きの配列 b[ ] を作成せよ (すなわち b["○"]=0, b["一"]=1, ... となる配列)。

課題 9-2. a[1]=1, a[2]=2, ..., a[10]=10 とした配列を、「for (j in a) print a[j]」で出力してみるとどういう順に出力されるか確認せよ。

課題 9-3. 1 列目に日付、2 列目に商品名、3 列目にその商品の単価、4 列目にはその商品の売り上げ個数、が並んでいるデータに対し、各商品毎に、商品名、その商品の総売り上げ個数と総売り上げ金額を表示するような AWK スクリプト kadai9-3.awk を作成せよ。

多次元配列を使いたい場合は、複数の添字をカンマ (,) で区切ればよい。例:

```
c[0, 0]=1;
for (n=1; n<=5; n++) for (k=0; k<=n; k++)
    c[n, k] = c[n-1, k-1] + c[n-1, k]; # 二項係数
for (k=0; k<=5; k++)
    printf "c[5, %d] = %d¥n", k, c[5, k];
```

a["新潟", "人口"] のような文字列の添字の 2 次元配列も利用できるが、多次元配列は、実際にはその添字文字列を連結 (カンマは 0x1C に変換) した文字列を添字とする 1 次元配列として実現しているだけなので、for 文で要素を参照する場合は、1 次元配列の場合と同様に一つの for で「for (j in a) print a[j]」のように行う。

課題 9-4. kamoku["月", 3] のような kamoku[<曜日>, <時限>] の 2 次元配列に、自分の履習科目名をセットし、それを for 文ですべて表示する AWK スクリプト kadai9-4.awk を作成せよ。

課題 9-5. 各行に 2 列あるデータに対して、その転置を出力する AWK スクリプト kadai9-5.awk を作成せよ。すなわち出力の 1 行目には、元のデータの 1 列目のデータが横に並び、出力の 2 行目には、元のデータの 2 列目のデータが横に並ぶようにする。

規則的な連想配列を作成する場合、split() と for 文を組み合わせると楽にできることがある。例えば、

```
split("日 月 火 水 木 金 土", weJa);
split("Sun Mon Tue Wed Thu Fri Sat", weEn);
for (j=1; j<=7; j++) weJ2E[weJa[j]] = weEn[j];
```

は、「weJ2E["日"]="Sun"」のような連想配列を作成する。

課題 9-6. split() と for 文を使って、h[0]=0, h[1]=1, . . . , h[9]=9, h["a"]=10, . . . , h["f"]=15 のような配列 (16 進数変換用) の初期化をできるだけスムーズに行う AWK スクリプト kadai9-6.awk を作成せよ。

課題 9-7. dates["January"]=31, dates["February"]=28 のように、英語の月名から日数を得るための配列の初期化をできるだけスムーズに行う AWK スクリプト kadai9-7.awk を作成せよ。

2 処理の中断

処理の途中でそれを中断する命令に next, exit がある。

- next : そこで現在の入力行に対する処理をやめて、先頭に戻って次の行の処理に進む
- exit : 処理をすべて終了し、行の読み込みもやめて、END ブロックがあればそこへ進む

例えば

```
NR == 1 { prev = $1 } # 1 行目
NR > 1 { print $1 - prev; prev = $1 } # 2 行目以降
```

は、

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

と書いても、

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

と書いても同じことになる。

ちなみにこれらのスクリプトは、1 行目では 1 列目の値を変数 prev に保存し、それ以降の行では 1 列目の値から prev の値を引いたものを出力し、再び 1 列目の値を変数 prev に保存しているので、結局元のデータの各行の 1 列目から前の行の 1 列目の差の値 (差分) を出力することになっている。

exit は、データにエラーがあった場合などに用いることが多いが、これらの他にも、ループ内での処理を中断する continue, break も C 言語と同様に使用できる[2]。

課題 9-8. 各行に 2 列の数値が並んでいるデータファイルの 2 行目以降に対して、1 列目の値と、(2 列目の値) − (前の行の 1 列目の値) の 2 列のデータを出力する AWK スクリプト kadai9-8.awk を作成せよ。

3 フィールドセパレータの変更

AWK のフィールドセパレータ (フィールドの区切り) は、デフォルトでは空白、またはタブになっているが、これはシステム変数 FS か、または gawk の実行時オプション -F に区切り文字の文字列を設定することで変更できる[3]。例えば、

```
Z:> gawk -F, "{print $2}" data
Z:> gawk "BEGIN{FS=¥",¥"}{print $2}" data
```

はいずれも、カンマ (,) で区切られたデータの 2 列目を出力するようになる。

表計算ソフト (MS-Excel, OpenOffice Calc 等) の多くは「CSV 形式」(カンマ区切りのテキスト形式) をサポートしているので、表計算データをこの形式で出力すれば、AWK のフィールドセパレータをカンマにすることで AWK でも表計算データを扱えるようになる。

課題 9-9. CSV 形式のデータの 2 行目以降のデータの 2 列目と 3 列目のそれぞれの平均値を求める AWK スクリプト kadai9-9.awk を作成せよ。

[2] next と exit は、それぞれ「行読み込み」というループに関する continue と break に対応する、と見ることができる。

[3] 正確に言えば、ここで指定するのは単純な文字列ではなく正規表現。

課題 9-10. CSV 形式のデータの 2 行目以降のデータの 1 列目が「男」である行に対して、2 列目と 3 列目のそれぞれの平均値を求める AWK スクリプト kadai9-10.awk を作成せよ。

課題 9-11. $x = 0.0$ から 0.1 刻みで $x = 10.0$ までの x と $\sqrt{x}$ の値を、各行カンマ (,) 区切りで 2 列に並べた 101 行のデータを作成する AWK スクリプト kadai9-11.awk を作成し、実際にそのデータを作成して kadai9-11.csv として保存し、それを表計算ソフトで読み込んでみよ。

4 リダイレクトとパイプ

AWK の `print` 関数、`printf` 関数は通常は出力を画面に流すが、AWK 内部からその出力をリダイレクトしてファイルに落とすことができる。

- `print ... > "fname"` : `print` コマンドの出力を fname という名前のファイルに出力する (上書き)。
- `print ... >> "fname"` : `print` コマンドの出力を fname という名前のファイルに出力する (追加出力)。
- `print ... | "cname"` : `print` コマンドの出力を cname という名前のコマンドにパイプ渡しする。

リダイレクションやパイプの記号の使い方は、いずれもバッチファイルと同じ形式になっている。`printf` も同じ形式で利用できる。ファイルやパイプコマンドへの出力が終了したときにファイルを閉じる `close` 関数も用意されている[4]。

- `close(s)` : ファイル名 (またはパイプコマンド名) s のファイルを閉じる。なお、s はリダイレクト先 (またはパイプ先) として指定したのと同じ文字列でなければいけない。

例えば、各行が学籍番号 (20XXYZNNN) で始まるデータファイルに対して、

```
/^20...1/ { print > "data-m.dat" }
/^20...2/ { print > "data-i.dat" }
/^20...3/ { print > "data-e.dat" }
/^20...4/ { print > "data-a.dat" }
```

[4]同時に開けるファイル数には制限があるので、たくさんのファイルへの出力を行う場合は、不要なファイルを順次 `close()` するとよい。

というスクリプトを実行すると、学籍番号の 6 桁目 (学科) が 1 のデータを data-m.dat に、2 のデータを data-i.dat に、という形でそれぞれのファイルに出力する。

課題 9-12. データの第 2 フィールドが「男」か「女」(カッコはついていない) である 4 列の入力データに対して、男の場合は data1.dat, 女の場合は data2.dat に第 1, 3, 4 フィールドのデータを出力する AWK スクリプト kadai9-12.awk を作成せよ。

課題 9-13. 10000 行ある入力データを、頭から 1000 行ずつ file01.dat, file02.dat, . . . , file10.dat という 10 個のファイルに出力する AWK スクリプト kadai9-13.awk を作成せよ。

5 AWK 変数を外部から指定

AWK スクリプト内部で使用する変数の初期値は、スクリプトを修正しなくても gawk の -v オプションで指定できる[5]。

- gawk -v [変数名]=[値] : スクリプトの実行前に [変数名] の変数に初期値として [値] を代入する

例えば、

```
{
    if (even == 1 && NR%2 == 0) print; # 偶数行を出力
    if (even == 0 && NR%2 == 1) print; # 奇数行を出力
}
```

というスクリプトをそのまま実行すると、even の初期値は 0 だから、入力データの奇数番目の行のみが出力されることになるが、このスクリプト (test1.awk とする) を

```
Z:> gawk -v even=1 test1.awk data
```

として実行すると、偶数番目の行が出力されるようになる。

この even のデフォルト値を 1 にしたければ、-v で指定しない場合の変数の初期値は空文字列なので、

[5]環境変数とシステム配列 ENVIRON[] を使って渡す方法もあるが、ここでは省略する。

```
BEGIN { if (even == "") even = 1 } # デフォルト値
{
    if (even == 1 && NR%2 == 0) print; # 偶数行を出力
    if (even == 0 && NR%2 == 1) print; # 奇数行を出力
}
```

のようにすればよい。

課題 9-14. 入力データの行のうち、列の数が nums 以上である行を出力する AWK スクリプト kadai9-14.awk を作成せよ。なお、nums のデフォルト値は 2 とし、-v で実行時に変更ができるようにせよ。

課題 9-15. 入力データの col 番目の列の平均値を求める AWK スクリプト kadai9-15.awk を作成せよ。なお、col のデフォルト値は 1 とし、-v で実行時に変更ができるようにせよ。

6 バッチファイルとの連携

バッチファイルの構文や変数、コマンドには色々な点で制限があるが、それらを補うのに AWK を利用することができる。ここではそれらの手法をいくつか紹介する。

まず、AWK に用意されている system() 関数を用いる方法がある。

- system(s) : AWK の外のコマンドプロンプトで文字列 s の内容を実行する

例えば、入力データに URL が 1 行ずつ書かれている場合、それを

```
{ system("start " $0) }
```

という AWK スクリプトにかければ、"start " と $0 (= 各行の URL) を連結した文字列、すなわち "start [URL]" を AWK の外で実行して AWK 内部に戻るので、各 URL をブラウザに表示することになる[6]。

system() に渡す文字列はバッチコマンドと同じもの (1 行) が指定できるので、「system("dir C:¥¥> root.dat")」のようにリダイレクションつきのコマンドや、パイプによる複数のコマンド列を並べることも可能である。

[6]先にブラウザを立ち上げておかないとうまくいかないかもしれない。

課題 9-16. 入力データの各行に 1 つずつディレクトリ名が書かれているとき、そのそれぞれに対して `dir /-c` コマンドを実行して、その出力全体を `dir-YYYYMMDD.dat` というファイル名 (YYYYMMDD は 20120415 のような今日の日付文字列) に保存する AWK スクリプト kadai9-16.awk を作成せよ。

バッチファイル内で AWK を使って、別のバッチファイルを作成し、そのバッチファイルを実行する、という方法もある。例えば、

```
@echo off
rem (1) 1 つ目のパラメータを使って子バッチを作成
gawk -v para=%1 -f mk.awk > tmp.bat
rem (2) 2 つ目, 3 つ目のパラメータを使って子バッチを実行
call tmp.bat %2 %3
rem (3) 子バッチを削除
del tmp.bat
```

のようにする方法である。ただ、この方法は、一つの作業のためにバッチファイルと AWK スクリプトの 2 つのファイルを保守/更新しなければいけないのが難点である。

課題 9-17. 入力データの各行に 1 つずつディレクトリ名が書かれているとき、そのそれぞれに対して `dir /-c` コマンドを実行して、その出力を `dir-YYYYMMDD.dat` というファイル名 (YYYYMMDD は 20120415 のような今日の日付文字列) にリダイレクト出力で保存するようなバッチファイルを作成する AWK スクリプト kadai9-17.awk を作成せよ。

コラム: フリーソフトの文化

多くの人は「フリーソフト」という言葉を聞いたことがあるだろうが、実は「フリーソフト」にも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は、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)」と呼ばれることが多い。

OSS のソフトは、以前は個人や趣味で集った数名が作成したものもあったが、現在は会社や団体で作成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) 上で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盛んに開発されている。

OSS のメリットは、

- 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
- ソーソが公開されているので不具合の改変も随時行われる
- 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
- ソースを見て勉強することができる (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)

などがある。

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。GNU は、世界的に有名なハッカー (悪い意味ではない) 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あり、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、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ていた。

ストールマンの当初の目標とはやや異なる形ではあるが、現在 GNU プロジェクトは OS を含む非常に多様なソフトを含むものになっている。gawk (GNU awk) も GPL に従う OSS の一つである。