2016 年 07 月 01 日

計算機実習 III (2016 年度)
第 10 回: AWK その 4

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

目 次

1 文字列の比較

AWK の条件式での文字列の比較は、数値の比較と同じ「==」「!=」(等しい、等しくない)、および「>」「<」「>=」「<=」が使える。その大小は辞書式順序 (コマンドプロンプトとは違い、正式な辞書式順序) で比較されるが、文字列と数値の比較を行った場合は自動的に数値が文字列に変換されて比較される。辞書式順序では、各文字の大小は ASCII コード順で、

```
"0" < "9" < "A" < "Z" < "a" < "z"
```

のようになっている。辞書式順の文字列比較は、例えば以下のようになる:

```
"30000" < "400" < "6" < "A0" < "AB" < "Ab" < "aB" < "ab"
```

これら以外にも文字列の条件文には、表 1 のマッチ演算子を用いることができる。

条件式	意味
s ~ /正規表現/	文字列 s が正規表現にマッチすれば真
s !~ /正規表現/	上の否定 (マッチしなければ真)

表 1: ~ 演算子とその否定

「正規表現」とは、詳しくは 2 節で説明するが、メタ文字がなければ普通の文字列と同じで、「正規表現にマッチする」とは、基本的には「正規表現が表す文字列を含む」ことを意味する。正規表現は、" " ではなく / / で囲んで指定する。「==」「!=」の場合と比較して書くと、例えば以下のようになる。

条件式の例	意味
s == "ABc"	文字列 s が "ABc" という文字列に等しいと真
s != "ABc"	文字列 s が "ABc" という文字列に等しくないと真
s ~ /ABc/	文字列 s が "ABc" という文字列を含むと真
s !~ /ABc/	文字列 s が "ABc" という文字列を含まないと真

例えば、スクリプト

```
{ if ($1 ~ /2013/) print $3 }
```

は、データファイルの 1 列目に "2013" が含まれている行の 3 列目を出力する。また、実行ブロックを略して条件だけを書くと、条件に合う行そのものを出力するので、

```
$3 ~ /<DIR>/
```

というスクリプトは、データファイルの 3 列目が "<DIR>" にマッチする行を出力する。

以下のように左辺の文字列とマッチ演算子を省略した場合は入力行全体 ($0) が対象となる。

条件式の例	意味
/正規表現/	「$0 ~ /正規表現/」と同じ
!/正規表現/	「$0 !~ /正規表現/」と同じ

例えば、「if (/2015/)」と「if ($0 ~ /2015/)」は同じ意味になるし、スクリプト

```
/wihle/ { print NR, $0 }
```

は「wihle」という文字列が含まれる行を行番号つきで表示する。これにより、ソースコード中で「while」を 「wihle」と間違えて書いた行を一度に探すことができる。

2 正規表現 その 1

/ / で囲む **正規表現** (regular expression) 文字列が表 2 の左列の文字 (**メタ文字**) を含む場合、その文字は右列のような特別な意味を持ち、それによりひとつの正規表現で

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 2: メタ文字一覧

複数の文字列のパターンを表現できるようになる。今回は、表 2 のうち、前半 7 つのメタ文字の使い方を説明し、残り 4 つのメタ文字は次回説明する。

まず、「.」「?」「*」「+」の例を示す。

正規表現の例	意味
2.3	"213", "2a3", "2X3", "2 と 3" などを表す
12?3	"13", "123" のいずれか
1?2?3	"3", "23", "13", "123" のいずれか
12*3	"13", "123", "1223", "12223" など
12+3	"123", "1223", "12223" など (上の "13" 以外)
12.*3	"12" で始まり 3 で終わる任意の文字列

表 2 の 1 つ目のメタ文字「.」は、どんな 1 文字ともマッチする。最初の例にあるように、全角文字 1 文字も「.」1 文字にマッチする。

表 2 の 2 つ目から 4 つ目の「?」「*」「+」は、その直前の文字の繰り返しを意味するので、単独では使えず、この前に文字 (を示すもの) がついていなければいけない。

メタ文字 ?, *, + が意味する繰り返しの回数は、? は 0 回か 1 回、すなわち「その直前の文字がない状態かある状態」を意味し、* は 0 回以上すべてを意味し、+ は 1 回以上すべてを意味する。よって、直前の文字の最小何個、最大何個の繰り返しなのか、によって後ろにつけるメタ文字を使い分けることになる (表 3)。

「?」「*」はコマンドプロンプトのワイルドカードに似ているが、ワイルドカードの方はそれ自体が (任意の) 文字を意味するので、AWK のメタ文字とはだいぶ異なること

	最小 0 個	最小 1 個
最大 1 個	?	(何もつけない)
最大の制限なし	*	+

表 3: ?,*,+ の使い分け

に注意。例えばワイルドカードでは「3???」は「3 から始まる 4 文字以下の文字列」を意味するが、正規表現では「3???」は「3?」と同じであり、?, *, + 自体を繰り返すことは正規表現では意味がない。

また、「/2*/」や「/2?/」という条件式は、その 0 個である「空文字列」も意味し、それにマッチする文字列 (それを含む文字列) とはすべての文字列になってしまうので意味がない [1]。

「.*」という正規表現は、「任意の文字の 0 回以上の繰り返し」なので「任意の文字列」を意味するが、上の例のようにその前後に文字列を指定して使われる。例えば、「<.*>」は、< > で囲まれた文字列にマッチする。

表 2 の 5 つ目の ¥ (エスケープ文字) は、C 言語同様「¥n」(改行)、「¥t」(タブ文字) を表すのに使えるし、メタ文字の前につけてその特別な意味を消すのに使う。例えば、¥* はメタ文字ではなく * そのものを表し、/ / 内で文字として「/」を指定する場合は「¥/」とする必要がある。例えば、「¥[.*¥]」は、[ ] で囲まれた文字列にマッチするが、[と] はメタ文字なので、¥ をそれぞれ左につける必要がある。

表 2 の 6 つ目と 7 つ目の ^ と $ は、は文字列の先頭と末尾を表す。マッチの条件式 (/ /) は、基本的にはその正規表現が表す文字列を「含む」かどうかを判別するが、^, $ を使うことで、その正規表現で「始まる」文字列や、正規表現で「終わる」文字列、その正規表現に完全に等しい文字列などを指定できるようになる。以下に例を示す。

条件式の例	意味
s ~ /2015/	文字列 s が "2015" を含むとき真
s ~ /^2015/	文字列 s が "2015" で始まるとき真
s ~ /2015$/	文字列 s が "2015" で終わるとき真
s ~ /^2015$/	文字列 s が "2015" に等しいとき真
s ~ /^2.*5$/	文字列 s が 2 で始まり 5 で終わるとき真
s ~ /^2?$/	文字列 s が空文字列か "2" のとき真
s ~ /^$/	文字列 s が空文字列のとき真

^ と $ の両方を使うと、指定した正規表現を「含む」文字列ではなく、指定した正規

[1] しかも、実習室の gawk (version 3.1.5) だと、match() によるそのような正規表現のマッチング時の RLENGTH の設定にバグがあるよう。

表現に「完全に等しい」文字列であるかどうかのチェックができる。

わずか 5 文字のスクリプト

```
/^20/
```

は、データファイルの "20" から始まる行を出力し、同じく 5 文字のスクリプト

```
!/^#/
```

は、「#」が行頭にない行のみを出力するので、結果的に「#」が行頭にある行 (AWK のコメント行) を削除して出力することになる。同様に、空行を削除するには、

```
!/^$/
```

とすればよい。なお、「NF>0」だけのスクリプトでもほぼ同じことになるが、これと上のスクリプトとは、空白のみからなる行に対しては結果が異なる。

3 関数 match()

正規表現にマッチした部分文字列の情報を取り出す関数として match() が用意されている。

関数	意味
match(s, r)	文字列 s が正規表現 r にマッチしたらその部分文字列の先頭位置を返す (マッチしなかったら 0 を返す)

表 4: match()

match() はさらにマッチした情報をシステム変数 RSTART, RLENGTH にセットする。

システム変数	意味
RSTART	マッチした部分文字列の開始位置
RLENGTH	マッチした部分文字列の長さ

表 5: RSTART と RLENGTH

この RSTART の値は、match() の返り値と同じになる。

例えば、

```
BEGIN { s = "12ab34cd" ; match(s, /a./);
    print RSTART, RLENGTH }
```

とすると、"ab" という部分文字列にマッチするため、RSTART は 3 に、RLENGTH は 2 になる。

なお、match() は、正規表現にマッチする部分文字列が複数ありうる場合は、最初に現れて、最も長くなるものが対象となる。例えば、

```
BEGIN { s = "2ab2ab2ab"; match(s, /a.*2/);
    print RSTART, RLENGTH }
```

とすると a で始まり 2 で終わる文字列のうち、最初に現れて、最長のものである、2 文字目から 7 文字目までの "ab2ab2" という部分にマッチするので、RSTART は 2 に、RLENGTH は 6 になる。

RSTART, RLENGTH の値と substr() と組み合わせることで、容易に正規表現にマッチした部分文字列などを取り出すことができる。例えば、

```
BEGIN { s = "abc12345defghi"
    if (match(s, /123../)) {
        s1 = substr(s, 1, RSTART - 1)
        s2 = substr(s, RSTART, RLENGTH)
        s3 = substr(s, RSTART + RLENGTH) }
    printf "s1=%s, s2=%s, s3=%s¥n", s1, s2, s3 }
```

とすると、2 行目の match() により RSTART=4, RLENGTH=5 となり、よって

```
s1="abc", s2="12345", s3="defghi"
```

が表示される。なお、マッチしない場合は、match() は 0 (すなわち偽) を返すので、上のような if 文にすれば、そのブロックは実行されない。また、RSTART, RLENGTH の値は、次の match() が行われるまで保持される。

検索する文字列内に、探したい文字列が複数ある場合、match() は最初に見つかった文字列の情報だけを返すので、上の s3 にあたる残りの文字列の部分を繰り返し検索する必要がある。例えば、スクリプト

```
/。/ { s = $0 # 行の内容を s に一旦保存
    while (match(s, /。/)) {
        print substr(s, 1, RSTART + RLENGTH - 1) # 出力
        s = substr(s, RSTART + RLENGTH) # その後ろの文字列に
    } }
```

は、データファイルの各行 ($0) に対し、句点 (/。/) を探し、先頭から句点までの (RSTART + RLENGTH − 1) 文字分を print で出力し、それ以後の文字列を対象としてまた同じ検索を行う、という作業を、while 文で残りの部分がマッチしなくなるまで繰り返している。

なお、この実行ブロックの先頭に /。/ がついているが、これにより、「。」を含む行だけに対してブロック内を実行することになる。先頭の /。/ がなくても結果は同じであるが、これをつけることで /。/ を含まない行に対する無駄な作業をせずに済む[2]。

コラム: gawk の国際化

gawk のような外国生まれのフリーソフトの多くは日本語のような 2 バイト文字には対応していないことが多い。通常は、日本語に対応していないアプリケーションでは、以下のような問題がある。

- 日本語がメニューに出ない (GUI アプリケーションの場合)
- ヘルプやエラーメッセージが日本語で見られない
- データとして日本語が扱えない

そのようなソフトで日本語が使えるようにすることを「日本語対応」とか「日本語化」と呼ぶ。

さらに、より多くの国や民族の言語に対応することを目指すことを「国際化」(Internationalization) と呼び、「日本語化」のように特定の言語用にソフトを改良する方向を「地域化」(localization) と呼んで区別することもある。現在は、文字コードを国際的にある程度統一して扱える Unicode (UTF–8) という規格ができたこともあり、外国生まれのフリーソフトもかなり「国際化」が進んできている。

gawk もバージョン 3.1.5 位から国際化が行われているが、gawk のような CUI ツールの国際化は、主に扱うデータやエラーメッセージへの対応であり、gawk では 多バイト文字を英数字などと同じように「1 文字」とみなす方向で国際化が行われている。

[2]ただし、実行時間の変化などはほとんどなく、単なる気持の問題でしかない。

例えば、ASCII データとしては ”abc” と ”日本語” という文字列は、前者は 3 バイト、後者はシフト JIS のような 2 バイト文字コードでは 6 バイト (UTF–8 では 9 バイト) であるが、以前の国際化されていない gawk では、

```
length("abc") = 3, length("日本語") = 6,
substr("日本語", 3, 2) = "本"
```

であったのに対し、バージョン 3.1.5 等の最近の gawk では、

```
length("abc") = 3, length("日本語") = 3,
substr("日本語", 2, 1) = "本"
```

となる (`length(s)` = 文字列 s の長さ)。

出力の方も、

```
printf "%6s¥n", "日本"
```

というスクリプトの出力は、バージョン 3.1.5 では、「スペース 2 つ + 日本」が表示されていた (すなわち「日本」を 4 文字と見ている) が、バージョン 3.1.8 以降では、「スペース 4 つ + 日本」が表示される (すなわち「日本」を 2 文字と見ている) ように改良されている。

全角文字を 1 文字として扱う仕組みは、文字データを日本語か非日本語かを区別せずに扱え、特に 1 バイト文字と 2 バイト文字が混在している文字列の処理の場合には便利である。しかし一方で、等幅フォントで全角文字を英数半角文字の 2 倍の幅に揃えたい場合などにはこの仕様は逆に不都合であったりする。

フリーソフトのプロジェクトでは、そのような地域毎の事情を、欧米人が多い開発側に理解してもらうことは難しく、よって、開発メンバーに日本人が参加していないプロジェクトでは、日本人にとってはやや不都合な国際化が起こることもままあり、日本独自の日本語化 (地域化) が必要な場面もまだまだありそうである。