AWK の第一歩

小栗栖 修
金沢大学理学部計算科学科
ogurisu@lagendra.s.kanazawa-u.ac.jp

2001 年 1 月 8 日 (月): 第 1 版
2002 年 10 月 10 日 (木):第 1.1 版

この文書の目標は AWK を使ったことがない人が使い始めるきっかけになればというところにあります。C や Java よりも気楽にプログラミングが楽しめて、もちろん実用的なプログラムが書けるのが AWK や perl，ruby に代表されるスクリプト言語です。すでに web 上にも AWK の文書が日本語のものも含めてたくさんあります。本格的なマニュアル形態の文書も、A4 に印刷して数枚程度のものもあります。が、ちょっと使ってみようというには本格的な文書はしんどいし、A4 数枚程度のものは簡潔すぎて何ができるのかよくわからないと思います。それを補完できればなと思っています。もし、この文書以上のことをしたくなったら、もっと本格的な本を読んでください

そんなわけで、この文書はマニュアルではありませんし、説明していないこともたくさんあります。また、間違いもあると思います。この文書の記述を信用して被害を蒙ったとしても著者はなんら責任を負えません。間違いを見付けた場合は是非とも著者に連絡をお願いします。すみやかに訂正するつもりです。[1]

なお、この文書の著作権は小栗栖にありますが、リンクは御自由にどうぞ。まさかそんな人はいないと思いますが、この文書でお金を稼ごう (無理、無理:p) ということでもない限り好きに利用してかまいません[2]。

目 次

[1] 第 1 版から第 1.1 版への変更も、読者からの指摘への対応がきっかけでした。ありがとうございました。
[2] とは言え、個人以外での利用の際は、一言メールしてくださるととても嬉しい。強制ではありませんけど。

1 AWKって何するもん？

例えば、大学のコンピュータにあるファイルを自宅に持って帰りたい。でもちょっと大きいのが幾つかあるからフロッピー 1 枚に収まるだろうか?なんてことはありがちです[3]。とりあえず、 `ls -l` でファイルの大きさをチェックしてみましょう。

```
% ls -l
合計 101572
-r--r--r--   1 ogurisu  ogurisu   280396  5 月 20 日 1999 年 auctex-9.9p.tar.gz
-r--r--r--   1 ogurisu  ogurisu   169498  5 月 26 日 1999 年 cmail-2.59.13.tar.gz
-rw-r--r--   1 ogurisu  ogurisu  1319271 10 月 24 日  08:32 gnuplot_3.7.tar.gz
-rw-r--r--   1 ogurisu  ogurisu   822766  9 月 27 日  18:59 ruby-1.6.1.tar.gz
-r--r--r--   1 ogurisu  ogurisu    51458  7 月 15 日 1998 年 sh-text.tar.gz
-r--r--r--   1 ogurisu  ogurisu   257496  5 月 20 日 1999 年 yatex1.66.tar.gz
%
```

[3] 今時、自宅からインターネット越しに転送すればいいんだけど。回線が遅いってことにしておきましょう。

それぞれのファイルの大きさを合計して 1.4 メガバイト (約 1,400,000 バイト) を超えなければフロッピー 1 枚に収まるでしょう。そこで、電卓を取出して — 電卓プログラムを起動して — ファイルサイズを合計しましょう。

これって、なんかだか変だと思いませんか? ファイルサイズのデータはコンピュータの中にあるのに、それをわざわざ人間の手で電卓 (プログラム) に入力するんですか?

こんなとき AWK を知っていれば、次のようにコマンドライン一発で計算できます。2900885 バイトもあるので、全然フロッピーに収まりませんね。

```
% ls -l | gawk '{a+=$5;} END {print a;}'
2900885
%
```

パイプ | の後ろにある gawk が AWK のコマンドです[4]。そのあとに '{a+=$5;} END {print a;}' と、なんだか呪文のようなことを書いていますが、これが AWK のプログラムです。

このプログラムは次のようにファイルに保存して実行することもできます。ファイル名はなんでも良いので、例えば sum.awk としましょう。

```
#!/usr/local/bin/gawk -f
# sum.awk:
{
    a += $5;
}
END {
    print a;
}
```

そしてさきほどの gawk 'a+=$5; END print a;' を gawk -f sum.awk に置き換えて実行します[5]。

```
% ls -l | gawk -f sum.awk
2900885
%
```

AWK では、C や Fortran と違いコンパイルも不要です。ぱっと書いて、ぱっと実行する。もし間違っていても、ちゃっちゃと書きなおして、すぐに再実行できます。ls のような他のプログラムから出力される大量のデータを要約するなどの作業に適しています。単純だけど人間が手でやるには面倒くさい作業を肩代りさせる**使い棄てのプログラム**に便利です[6]。

また、$\sin(x)$ の値を区間 $[0, \pi]$ で 100 等分して出力したいなんて簡単な計算のときは、わざわざ C を使わなくても、次のようなスクリプト sin.awk を作って実行すれば十分です。

```
#!/usr/local/bin/gawk -f
# sin.awk:
BEGIN {
    for (i = 0; i <= 100; i++)
        printf("%1.6f\n", sin(i*3.1415/100));
}
```

[4] gawk は GNU(FSF 製) の AWK で、多くの Linux のディストリビューションに入っている AWK はこの gawk で、/usr/local/bin/か/usr/bin/にあることでしょう。

[5] 最初の gawk -f も省略する方法があります。後ほど紹介します。

[6] 書こうと思えばアセンブラだの複雑なプログラムも書けますし、実際にあるそうです。

```
% gawk -f ./sin.awk
```

書式が C に似ていると思いませんか? C を少し知っていれば新しく覚えることは極わずかです[7]。

なお、AWK という名前は作者の 3 人の頭文字を並べたもので、K は「プログラミング言語 C」の著者 Kerninghan です。A は名著「データ構造とアルゴリズム」の著者の一人 Aho、W は Weinberger です。情報科学の三巨人の共作ですね。

2 AWK は行単位で処理を行う

最初に覚えて欲しいのは、AWK は入力を行単位で一行づつ順番に処理していくということです。例として次のデータファイル miyabe.data を処理することにしてみます。このデータファイルは、宮部みゆきの文庫を一行あたり一冊づつ、書名、本体価格、発行日、発行社、文庫、備考、を**空白**で区切って記入してあります。ただし、この空白はいわゆる**半角の空白**です。タブでも構いません。なお、日本語コードは EUC にしておくのが無難です[8]。

```
書名　本体価格　発行　発行社　文庫　備考
火車　743　98/02/01　新潮社　新潮文庫 み-22-8
かまいたち　505　96/09/01　新潮社　新潮文庫 み-22-6　短編集
蒲生邸事件　1650　96/10/10　毎日新聞社
堪忍箱　1456　96/10/30　新人物往来社
クロスファイア　［上・下］　819　98/10/30　光文社　カッパ・ノベルス
幻色江戸ごよみ　552　98/09/01　新潮社　新潮文庫 み-22-9
心とろかすような マサの事件簿　1300　97/11/28　東京創元社　　短編集
コットン４　750　90/03/01　大陸書房　　発行社連絡不能
今夜は眠れない　552　98/11/18　中央公論社　中公文庫 み-32-1
淋しい狩人　466　97/02/01　新潮社　新潮文庫 み-22-7
ステップファザー・ステップ　563　96/07/15　講談社　講談社文庫 み-42-1
スナーク狩り　1068　92/06/10　光文社　カッパノベルスハード
地下街の雨　1165　94/04/01　集英社
天狗風　霊験お初捕物控〈二〉　1800　97/11/15　新人物往来社
東京下町殺人暮色　505　94/10/20　光文社　光文社文庫 み-13-1　東京殺人暮色改題
とり残されて　466　95/12/10　文藝春秋　文春文庫 み-17-2
長い長い殺人　819　97/05/25　光文社　カッパ・ノベルス
初ものがたり　553　97/03/17　ＰＨＰ研究所　ＰＨＰ文庫 み-14-1
鳩笛草　796　95/09/25　光文社　カッパノベルス
パーフェクト・ブルー　534　92/12/25　東京創元社　創元推理文庫 M-み-1-1
人質カノン　1359　96/01/30　文藝春秋
震える岩　霊験お初捕物控　695　97/09/15　講談社　講談社文庫 み-42-2
平成お徒歩日記　1500　98/06/30　新潮社
返事はいらない　427　94/12/01　新潮社　新潮文庫 み-22-3
本所深川ふしぎ草紙　427　95/09/01　新潮社　新潮文庫 み-22-5
魔術はささやく　505　93/01/25　新潮社　新潮文庫 み-22-1
```

[7] C を知らないとか C は苦手な人でも、AWK は簡単に使えます。念のため。
[8] ちゃんと日本語コードに対応した jgawk などもあります。

夢にも思わない　895　97/10/25　中央公論社　C☆NOVELS
理由　1200　98/06/01　朝日新聞社
龍は眠る　660　95/02/01　新潮社　新潮文庫 み-22-4
レベル7　699　93/09/25　新潮社　新潮文庫 み-22-2
我らが隣人の犯罪　369　93/01/10　文藝春秋　文春文庫 み-17-1

2.1　AWK は行を空白で分割して処理する

AWK は、入力から一行データを読むたびに空白文字を区切としてデータを分割し、最初から順番に変数 $1、$2、$3、... に代入します。入力のある一行がつぎのようなものだったとしましょう。

火車　743　98/02/01　新潮社　新潮文庫 み-22-8

この場合、$1=火車、$2=743、$3=98/02/01、$4=新潮社、$5=新潮文庫、$6=み-22-8、となります。
そこで、'{print $1;}' というプログラムを AWK に与えてやりましょう。すると、各行ごとにこのプログラムが実行されます。つまり$1 の内容である書名が印刷 (print) されます。実際にこの仕組を使って、書名だけを取出してみましょう。

```
% gawk '{print $1;}' miyabe.data
書名
火車
かまいたち
蒲生邸事件
堪忍箱
クロスファイア　［上・下］
幻色江戸ごよみ
心とろかすような
コットン４
今夜は眠れない
　　（省略）
```

データファイルの順に書名が取出せていますね。自分でもやってみてください。プログラムの前後を '{ と }' で囲っていますが、今はそういうものだと思っておいてください。
同じように本体価格を取出してみましょう。価格は$2 に納められているはずです。

```
% gawk '{print $2;}' miyabe.data
本体価格
743
505
1650
1456
819
552
マサの事件簿
750
552
```

（省略）

途中で「マサの事件簿」という予定外のものが出力されています。元データを良く見ると題名が「心とろかすような　マサの事件簿」となっています。

```
（省略）
クロスファイア　［上・下］　819　98/10/30　光文社　カッパ・ノベルス
幻色江戸ごよみ　552　98/09/01　新潮社　新潮文庫　み-22-9
心とろかすような マサの事件簿　1300　97/11/28　東京創元社　　短編集
（省略）
```

「マサの事件簿」という副題がついていて、題名の前後を半角の空白で区切ってあったため、この行では２つ目のデータが本体価格ではなく「マサの事件簿」になって、それが $2 に代入されてしまったのです。その２冊上に「クロスファイア　［上・下］」という本があるのですが、こちらは全角の空白で区切ってあるので、うまく処理されています。いまのところ、仕方ないのでこのような区切は**全角の空白**を使うことにしておきましょう。のちほど (第 2.6 節)、もっとましな対策を紹介します。

この「心とろかすような　マサの事件簿」のデータを修整したファイルを miyabe.new としておきます。

もうお分りかと思いますが、書名と出版日だけを見たいなら、$1 と$3 を print してやります。ただし、$1 と$3 をそのまま並べると次のように２つがひっついて表示されます。

```
% gawk '{print $1 $3;}' miyabe.new
書名発行
火車 98/02/01
かまいたち 96/09/01
蒲生邸事件 96/10/10
（省略）
```

$1 と$3 をカンマ , で区切っておくと、適当に空白を入れてくれます。

```
% gawk '{print $1, $3;}' miyabe.new
書名 発行
火車 98/02/01
かまいたち 96/09/01
蒲生邸事件 96/10/10
（省略）
```

2.2　AWK の組込変数 NR と NF

AWK によって分割された行のデータをそれぞれを**フィールド**と言います。またこのフィールドのひと固まりである各行を**レコード**と呼びます。この文庫のデータの場合、一冊の文庫全体のデータで一つのレコードをつくり、書名などの項目がフィールドになります。

AWK にはいくつか**組込変数**があって、そのうちの NR は幾つめのレコードを処理しているか、NF は処理中のレコードにいくつのフィールドがあるかを記憶してます。この NR を使って行番号を追加できます。

```
% gawk '{print NR, $1, $3;}' miyabe.new
1 書名 発行
```

```
2 火車 98/02/01
3 かまいたち 96/09/01
4 蒲生邸事件 96/10/10
    （省略）
```

$0 はレコード全体、つまり元のデータ行そのものとなりますから、次のようにすることで行番号を振ることもできます (cat -n miyabe.new と同じ)。

```
% gawk '{print NR, $0;}' miyabe.new
1 書名　本体価格　発行　発行社　文庫　備考
2 火車　743　98/02/01　新潮社　新潮文庫 み-22-8
3 かまいたち　505　96/09/01　新潮社　新潮文庫 み-22-6　短編集
4 蒲生邸事件　1650　96/10/10　毎日新聞社
    （省略）
```

NR を使って特定の行だけを取出すこともできます。次の例では NR が 3 に等しいところ、つまり 3 行目だけを取出しています。{　} で囲まれたプログラムの前に条件式を書いてやればいいのです。

```
% gawk 'NR == 3 {print NR, $0;}' miyabe.new
3 かまいたち　505　96/09/01　新潮社　新潮文庫 み-22-6　短編集
%
```

次の例では 1 行目から 3 行目を取出します。

```
% gawk 'NR <= 3 {print NR, $0;}' miyabe.new
1 書名　本体価格　発行　発行社　文庫　備考
2 火車　743　98/02/01　新潮社　新潮文庫 み-22-8
3 かまいたち　505　96/09/01　新潮社　新潮文庫 み-22-6　短編集
%
```

この条件は、&&や||、!を使って、論理和や論理積、否定を取ることができます。

```
% gawk 'NR >4 && NR < 7 {print NR, $0}' miyabe.new
5 堪忍箱　1456　96/10/30　新人物往来社
6 クロスファイア　［上・下］　819　98/10/30　光文社　カッパ・ノベルス
%

% gawk 'NR == 2 || NR == 4 {print NR, $0}' miyabe.new
2 火車　743　98/02/01　新潮社　新潮文庫 み-22-8
4 蒲生邸事件　1650　96/10/10　毎日新聞社
%
```

次の例は NR != 1 としても同じになります。

```
% gawk '!(NR == 1) {print NR, $0}' miyabe.new|head
2 火車　743　98/02/01　新潮社　新潮文庫 み-22-8
3 かまいたち　505　96/09/01　新潮社　新潮文庫 み-22-6　短編集
4 蒲生邸事件　1650　96/10/10　毎日新聞社
5 堪忍箱　1456　96/10/30　新人物往来社
    （省略）
```

NF の使用例はのちほど見ましょう。

2.3 AWK で特定の行だけを処理する

データファイル全体から特定のフィールドを取出すことをみてきましたが、「講談社から出版されたものだけ見たい！」ときもあります。そんなときは、プログラムの { の前に / で囲って絞りこみの条件となる文字列を書くと、その文字列を含む行だけが処理の対象になります[9]。

```
% gawk '/講談社/ {print $0;}' miyabe.new
ステップファザー・ステップ　563　96/07/15　講談社　講談社文庫 み-42-1
震える岩　霊験お初捕物控　695　97/09/15　講談社　講談社文庫 み-42-2
%
```

講談社のものだけが取出されていますね。

ところで、この例のように各行をまるごと表示するときは$0 を省略することができます。print は引数が指定されない場合、$0 を表示するように作られているのです。

```
% gawk '/講談社/ {print;}' miyabe.new
ステップファザー・ステップ　563　96/07/15　講談社　講談社文庫 み-42-1
震える岩　霊験お初捕物控　695　97/09/15　講談社　講談社文庫 み-42-2
%
```

'{print;}' も省略できます。/文字列/だけを与えると、そのパターンを含む行だけを表示します。

```
% gawk '/講談社/' miyabe.new
ステップファザー・ステップ　563　96/07/15　講談社　講談社文庫 み-42-1
震える岩　霊験お初捕物控　695　97/09/15　講談社　講談社文庫 み-42-2
%
```

同じように「96 年に出版されたものだけ見たい!」ときもあります。今の場合、絞り込みの条件を /96/ としましょう

```
% gawk '/96/' miyabe.new
かまいたち　505　96/09/01　新潮社　新潮文庫 み-22-6　短編集
蒲生邸事件　1650　96/10/10　毎日新聞社
堪忍箱　1456　96/10/30　新人物往来社
ステップファザー・ステップ　563　96/07/15　講談社　講談社文庫 み-42-1
鳩笛草　796　95/09/25　光文社　カッパノベルス
人質カノン　1359　96/01/30　文藝春秋
%
```

どの行にも 96 が含まれていますね。え、「鳩笛草は 95 年の出版じゃないか」って?これは価格が 796 で、そこに 96 が含まれていたから表示されちゃったんですね。AWK は 96 年という意味を考えて処理しているのではなく、96 という文字の並びが含まれるかどうかしかチェックしていないのです。

この場合、発行年月日を「年/月/日」という形式で保存してあるので、直後の / も含めて、次のようにしてやると良いでしょう。ただし、単純に /96// とすると、区切の / との区別が付かなくなりますから、 \ を直前に置いて /96\// とします。

[9] これは正規表現と呼ばれる文字列の指定方法の一番簡単な例です。正規表現 (第 2.4 節) の詳しいことはもっとあとで説明します。

```
% gawk '/96\//' miyabe.new
かまいたち　505　96/09/01　新潮社　新潮文庫 み-22-6　短編集
蒲生邸事件　1650　96/10/10　毎日新聞社
堪忍箱　1456　96/10/30　新人物往来社
ステップファザー・ステップ　563　96/07/15　講談社　講談社文庫 み-42-1
人質カノン　1359　96/01/30　文藝春秋
%
```

もし \ を置き忘れるとエラーになります。

```
% gawk '/96//' miyabe.new
gawk: cmd. line:1: /96//
gawk: cmd. line:1:     ^ syntax error
gawk: cmd. line:1: /96//
gawk: cmd. line:1:      ^ unexpected newline or end of string
%
```

この /文字列/ の条件指定に対しても、&&や||、!が使えます。

```
% gawk '/96\// || /97\//' miyabe.new
かまいたち　505　96/09/01　新潮社　新潮文庫 み-22-6　短編集
蒲生邸事件　1650　96/10/10　毎日新聞社
堪忍箱　1456　96/10/30　新人物往来社
心とろかすような　マサの事件簿　1300　97/11/28　東京創元社　　短編集
淋しい狩人　466　97/02/01　新潮社　新潮文庫 み-22-7
ステップファザー・ステップ　563　96/07/15　講談社　講談社文庫 み-42-1
天狗風　霊験お初捕物控〈二〉　1800　97/11/15　新人物往来社
長い長い殺人　819　97/05/25　光文社　カッパ・ノベルス
初ものがたり　553　97/03/17　ＰＨＰ研究所　ＰＨＰ文庫 み-14-1
人質カノン　1359　96/01/30　文藝春秋
　　（省略）
```

なお、NR を使った行の指定（第 2.2 節）でも次のように'{print;}' を省略できます。

```
% gawk 'NR == 2' miyabe.new
火車　743　98/02/01　新潮社　新潮文庫 み-22-8
%
```

2.4　AWK で正規表現

処理対象の行の指定に / で囲んだ文字列を使ってきましたが、これは正規表現という文字列のパターン指定方法の一番簡単な形です。もう少し正規表現のことを知ると、もっと多彩なテキスト操作ができるようになります。正規表現は、mule や vi、grep、sed など UNIX 系のツールで大抵使えます。

例を見ていきましょう。まず後で説明する幾つかの例外の文字を除いて

```
/abc123/
```

などアルファベットや数字などを並べたものは、並べられた文字そのものを表します。この例の場合、abc123 です。AWK では、

```
% gawk '/abc/' file
```

とすれば、file 内の abc を含む行が全て出力されました。このことを「/abc/にマッチした行が出力された」と言います[10]。
次に例外の文字を説明します。例外となるのは次の文字で**メタ文字**と呼ばれます。

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

これらの文字は以下に説明する特別な意味を持っています。

2.4.1 縦棒 |

```
/abc|def/
```

とすると「abc または def」となります。

```
% gawk '/abc|def/' file
```

とすると file 内の abc か def を含む行が全て出力されます。

```
/abc|def|0123/
```

などと繰り返すこともできます。

2.4.2 ピリオド .

改行以外の (タブやスペースなどの空白文字を含む) 任意の文字ひとつを表します。

```
/a...b/
```

は、3 つの文字が a と b で挟まれた文字列全てにマッチします。たとえば、aaaab、a012b、a　　b (空白が間に 3 つ) などです。

```
% gawk '/a...c/' file
```

とすれば、file 内の「3 つの文字が a と b で挟まれた文字列」を含む行が全て出力されます。

2.4.3 アスタリスク *

直前の文字の 0 回以上の繰り返しを表します。

```
/ab*c/
```

は、ac、abc、abbc、abbbc、abbbbc、、、にマッチします。プラス記号 + は、**直前の文字**の 1 回以上の繰り返しを表します。

[10] /abc/と書いたとき正規表現そのものは abc だけです。abc とだけ書くと正規表現である abc とただの文字列の abc との区別がつかないので、/で囲むのです。

```
/ab+c/
```

は、abc、abbc、abbbc、abbbbc、、、にマッチします。*と違って ac にマッチしないことに注意してください。疑問符 ？ は、**直前の文字**が 0 個か 1 個のときにマッチします。

```
/ab?c/
```

は、ac と abc にマッチします。ピリオド . とアスタリスク * とを組合わせると、

```
/a.*c/
```

は a と c に挟まれた 0 個以上の文字がある文字列にマッチします。

2.4.4 カレー ^ とダラー $

カレー ^ とダラー $ は、それぞれ行頭と行末にマッチします。

```
/^abc/
```

は、最初の 3 文字が abc である文字列にマッチします。たとえば"abcdef"にはマッチしますが、"0abcdef"にはマッチしません。

```
/abc$/
```

は、最後の 3 文字が abc である文字列にマッチします。たとえば"0123abc"にはマッチしますが、"abcdef"にはマッチしません。

```
/^$/
```

とすると一切文字の含まれない空行にマッチします。!を使って否定して

```
% gawk '!/^$/' file
```

とすると、file から空行を削除したものが出力されます。
ここで、ちょっとだけ実用的な例をお見せしておきます。ワークステーションでは各人のメールは (アカウントが abcd0123 なら) ファイル/var/mail/abcd0123 に保存されています。

```
% gawk '/^From/' /var/mail/abcd0123
```

とすると、From から始まる行だけが表示されるので誰からメールが届いているかを調べることができます。

2.4.5 文字クラス [ ]

[ ] を使うと文字の集合を決めることができます。この集合を文字クラスと言います。例えば、

```
/199[0123456789]/
```

とすると、1990、1991、1992、1993、1994、1995、1996、1997、1998、1999、にマッチします。このような連続した文字の場合、

```
/199[0-9]/
```

と略記できます。

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

とすれば 1 以上の整数を表わす文字列にマッチします。ひとつだけ注意が必要なのは [ ] の中に^を使うときです。もし次のように、^が [ ] 内の最初にある場合、

```
/[^0-9]/
```

は、0、1、2、3、4、5、6、7、8、9、以外の文字を表します。

```
/a[^ ]+b/
```

とすると、a と b の間に空白以外の文字が 1 個ある文字列にマッチします。文字クラスの中に^を入れたいときは 2 文字目以降に入れましょう。

```
/[a-z^]/
```

は、英小文字と^のいずれか一文字にマッチします。

2.4.6 グルーピング ( )

( ) は、グルーピングに使います。

```
/ab(cd)*/
```

は、ab、abcd、abcdcd、abcdcdcd、... にマッチします。また、次のように | と組合わせると

```
/(a|b)cd/
```

は acd と bcd にマッチします。

```
/((a|b)?cd|def)/
```

とすれば、cd か、acd か、bcd か、def にマッチします。

2.4.7 エスケープ文字 \

さて最後に問題があります。今、説明したメタ文字、

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

これらの文字そのものを指定したい場合はどうすれば良いのでしょうか？つまりピリオドを指定しているつもりで

```
/./
```

と書くと、改行以外の全ての文字にマッチしてしまいます。こんなときには、最後に残った \ を使います。これはエスケープ文字と呼ばれていて、メタ文字の特殊な働きをキャンセルする働きがあります。

```
/\./
```

でピリオドそのものだけに一致するようになります。\ もメタ文字の一つですから、\ に一致させたいときは

```
/\\/
```

とします。

メタ文字	意味
|	論理和
.	改行以外の任意の一文字にマッチ
*	直前の文字の 0 回以上の繰り返しにマッチ
+	直前の文字の 1 回以上の繰り返しにマッチ
?	直前の文字がない、または 1 個あることにマッチ
^	行頭、または文字列の先頭にマッチ
$	行末、または文字列の終端にマッチ
[ ]	文字クラス
( )	グルーピング
\	エスケープ文字

2.4.8 エスケープシーケンス

エスケープ文字 \ には、もう一つの使い方があります。次のような不可視の文字を表現します。この表記をエスケープシーケンスと言い、正規表現内でも用いることができます。C と同じですし、タブ \t と改行 \n を覚えておけば、まず事足りるでしょう。空白文字にマッチさせるには、まじめに書けば

```
/[ \t\f\n\r\v]/
```

となりますが、AWK で通常のテキストを処理するには

```
/[ \t]/
```

で十分でしょう。

\\	バックスラッシュそのもの
\a	「警告」文字。通常は ASCII BEL 文字である。
\b	バックスペース
\f	改ページ
\n	改行
\r	復帰 (キャリッジリターン)
\t	水平タブ
\v	垂直タブ
\x[0-9a-f]+	\x に続く 16 進数で表現された文字。"\x41"は大文字の A である。
\ddd	1 桁か 2 桁か 3 桁の 8 進数で表現された文字。"\101"は大文字の A である。
\c	文字 c そのもの。

2.5 AWK で入力全体を集計する

行単位の処理を見てきましたが、たとえば「これらの文庫を全部買ったら、いくらになるのだろう？」という疑問には、各行にある価格の総和を求めなければなりませんから行単位の処理だけではすみません。そこで、BEGIN と END というパターンが必要になります。

```
% gawk 'BEGIN {処理 1} {行単位の処理} END {処理 2}' miyabe.new
```

の形式でプログラムを与えると、AWK は、1 行目を処理する前にまず**処理** 1 を行い、その後で**行単位の処理**を入力が終るまで繰り返し、最終行に対する**行単位の処理**の終ってから最後に**処理** 2 を行います。BEGIN や END は一方または両方を省略しても構いません。

```
% gawk '{行単位の処理} END {処理 2}' miyabe.new
% gawk 'BEGIN {処理 1} {行単位の処理}' miyabe.new
% gawk '{行単位の処理}' miyabe.new
```

また「行単位の処理」を/正規表現/や NR を使って特定の行だけに行わせても構いません。

```
% gawk '/96\// {行単位の処理} END {処理 2}' miyabe.new
% gawk 'BEGIN {処理 1} /96\// {行単位の処理}' miyabe.new
% gawk '/96\// {行単位の処理}' miyabe.new
```

実例を上げましょう。miyabe.new について「これらの文庫を全部買ったら、いくらになるのだろう?」という疑問には、**変数**all に価格の合計を保存することにして、次のようにします。変数名はアルファベットの並びならなんでも構いません。

```
% gawk 'BEGIN {all=0;} { all += $2; } END { print all;}' miyabe.new
25798
%
```

まず BEGIN で**変数**all を 0 に初期化しています。その後、各行の処理で all に$2 の内容 (一冊づつの値段) を加算しています。 += という演算子は C でお馴染の加算用の演算子ですね。そして全ての行の処理が終ったとき、全ての価格を加算したものが all に入っていますので、END の処理で総和 all を表示しているのです。消費税を考慮に入れたいなら 1.05 を掛けておけばいいですね。

```
% gawk 'BEGIN {all=0;} { all += $2; } END { print all*1.05;}' miyabe.new
27087.9
%
```

96 年の出版分だけなら/正規表現/を使って和を取る部分を制限しましょう。

```
% gawk 'BEGIN {all=0;} /96\// { all += $2; } END { print all*1.05;}' miyabe.new
5809.65
%
```

ここでは説明のために BEGIN を使って変数を 0 に初期化しましたが、実は AWK では初めて使われる変数は 0 に自動的に初期化されますので、ここで出てきたプログラムは全部 BEGIN {all=0;} を省略できます。

```
% gawk '/96\// { all += $2; } END { print all*1.05;}' miyabe.new
5809.65
%
```

2.6 AWK のフィールドの区切を変更する

最初のほうの例で、「心とろかすような　マサの事件簿」という書名の元データが半角の空白で副題が区切られていて、処理に失敗したのを覚えていますか?　あのときは、このような区切は全角の空白を使うことにすると、データファイルの作り方を制限しました。しかし、人の目には空白文字の種類の違いは非常に分りにくいので、良い解決方法ではありません。

そこで区切文字のほうを半角の空白から他の文字へ変えてしまいましょう。ここではカンマ　,　に変更してみます。まず、データファイルの区切を　,　に変更したものを作り、そのファイル名を miyabe.csv とします。

AWK プログラムのほうは、オプション-F の直後に変更したい区切文字　,　を書きます。

```
% gawk -F , '{ print $1, $3; }' miyabe.csv
書名 発行
火車 98/02/01
かまいたち 96/09/01
蒲生邸事件 96/10/10
堪忍箱 96/10/30
クロスファイア［上・下］ 98/10/30
幻色江戸ごよみ 98/09/01
心とろかすような マサの事件簿 97/11/28
コットン　4 90/03/01
今夜は眠れない 98/11/18
    （省略）
%
```

オプション-F を使わずに、BEGIN を使って区切文字 (Field Separator と呼びます) を表わす組込変数 FS に変更したい文字を代入しておいても構いません。

```
% gawk 'BEGIN { FS=","; } { print $1, $3; }' miyabe.csv
```

こちらの使い方はスクリプト（第 3 節）で良く使われます。

ところで、このように一つのレコードの要素を一行に書き、各要素をカンマ (,) で区切って保存する形式を CSV (comma-separated value) と言って良く使われています。市販の表計算ソフトなどでもデータをこの形式で保存できることが多いそうです[11]。

3　AWK のスクリプトを書く

ここまでは AWK のプログラムをコマンドラインに直接書いて実行してきましたが、このプログラムをファイルに保存しておいて実行することもできます。

3.1 AWK スクリプトの簡単な例

最初の例として、次のコマンドラインと同じ処理をするスクリプトを作って使ってみましょう。

```
% gawk 'BEGIN { FS=","; } /96\// { print $1, $3; }' miyabe.csv
```

[11] 多いそうですと伝聞調なのは、著者が久しく市販のソフトを使っておらず、本当に伝聞だからです。

次の内容を fs.awk に保存しておきます。ほとんどコマンドラインのプログラムを適当に改行して書きなおしただけですね。

```
#!/usr/local/bin/gawk -f
# fs.awk: 区切文字を空白からカンマ , に変更する例
BEGIN {
    FS = ",";
}

/96\// {
    print $1, $3;
}
```

そして、gawk のオプション-f の直後にこのスクリプトを指定して、次のように実行します。

```
% gawk -f ./fs.awk miyabe.csv
かまいたち 96/09/01
蒲生邸事件 96/10/10
堪忍箱 96/10/30
ステップファザー・ステップ 96/07/15
人質カノン 96/01/30
%
```

このように AWK のプログラムが保存されたファイルのことを**スクリプト**と呼びます。AWK プログラムと呼んも良いのですが、AWK のようにコンパイル作業なしで実行できるような言語のプログラムはスクリプトと呼ばれることが多いようです。また、拡張子に.awk をつけましたが、これも単なる慣習です。しかし、mule/emacs を使っているのなら、この拡張子を付けておくと AWK 用のモードに自動的になるので便利でしょう[12]。

なお次のように改行せずにスクリプトを書いても全く同様に動きますが、こんな書き方をしたのではプログラムの構造もはっきりせず、良いことは一つもありません。構造がはっきり分るように行を変えてインデントをつけて書きましょう。

```
#!/usr/local/bin/awk -f
# fs.0.awk: 良くない書式の例
BEGIN { FS = ","; } /96\// { print $1, $3; }
```

3.2 AWK スクリプト自体をコマンドにする

スクリプト fs.awk の 1、2 行目を見てください。コマンドラインのプログラムではなかったものですね。

```
#!/usr/local/bin/gawk -f
# fs.awk: 区切文字を空白からカンマ , に変更する例
BEGIN {
    FS = ",";
}
```

[12] 自動的に AWK 用のモードにならない場合は M-x awk-mode をタイプします。

```
/96\// {
    print $1, $3;
}
```

行中に#がある場合、そこから後ろはコメントとなり、プログラムの実行とは関係がなくなります。行の途中からコメントにすることもできます。適切に最低限必要なコメントを書くようにしましょう。

しかし、この説明では 1 行目の意味は分りませんよね。これは次のように使われます。まず、chmod コマンドを使ってスクリプトに実行属性を与えます。

```
% chmod 755 fs.awk
% ls -l fs.awk
-rwxr-xr-x    1 ogurisu  ogurisu       120 Dec 10 02:18 fs.awk
%
```

chmod +x fs.awk としても実行属性を与えることができます。この実行属性は、この作業を一度行なえばそれ以降はずっと有効になります。こうしておくと、次のようにスクリプトファイル名を指定するだけで、awk のスクリプト fs.awk が実行できます。

```
% ./fs.awk sample.cvs
```

UNIX 系の OS では、ファイルの最初の 2 バイトが#!で始まる場合、そのファイル自身を引数として#!に続くファイル名 (この場合/usr/local/bin/gawk -f) を実際の実行コマンドとして、実行する機能があるのです。[13]つまり、次の 2 つは同等なのです。

```
% ./fs.awk sample.cvs
```

```
% /usr/local/bin/gawk -f ./fs.awk sample.cvs
```

これは UNIX 自身がそういう仕組を持っているとしか説明のしようがありませんが、とても便利です。まず、自分のホームディレクトリに bin というディレクトリを作っておきます。

```
% mkdir ~/bin
```

そして、~/.cshrc に次の一行を追加して、ログインしなおしましょう。

```
set path = (~/bin $path)
```

bash のユーザなら、次の一行を~/.profile か~/.bashrc に追加します。

```
PATH=~/bin:$PATH
```

そして、役に立つスクリプトができたら、chmod で実行属性を与えて、どんどん~/bin にコピーしておくのです。すると、いつでもどこでもそのスクリプトが普通のコマンドと同じように使えます。

[13] もし gawk が/usr/local/bin 以外のところにある場合は、それに合わせて/usr/bin/gawk などとしてください。

3.3 AWK におけるパターンとアクション

AWK のプログラムは、パターンとアクションの組の列と (もし必要なら) 関数定義から成ります。関数定義の説明は後 (第 3.8 節) にして、ここではパターンとアクションの説明をしましょう。

AWK のプログラムにおけるパターンとアクションの組は、プログラム中に次の形式で記述されます。

```
pattern   { action statements }
```

パターンには BEGIN と END、/正規表現/や NR を使った行指定などがあります。アクションは、そのパターンに入力行がマッチしたときに実際に行われる処理の記述です。例えば、

```
/96\// { print $1, $3; }
```

は /96/ がパターンで、{ print $1, $3; } がアクションです。

```
% gawk '{print $1, $3; }' miyabe.new
```

としたときはパターンの指定がないので、この場合は全ての入力行にマッチするとされます。また

```
% gawk '/96\//' miyabe.new
```

とアクションの指定がない場合はデフォルトのアクションとして '{print;}' が使われます。

実は今までの例では、パターンとアクションの組が一つの例だけを示してきましたが、 ; で区切って複数指定しても構いません。次のようにすれば、95 年と 96 年の出版物が表示できます。

```
% gawk '/95\//; /96\//' miyabe.new
かまいたち  505  96/09/01  新潮社  新潮文庫 み-22-6  短編集
蒲生邸事件  1650  96/10/10  毎日新聞社
堪忍箱  1456  96/10/30  新人物往来社
ステップファザー・ステップ  563  96/07/15  講談社  講談社文庫 み-42-1
とり残されて  466  95/12/10  文藝春秋  文春文庫 み-17-2
鳩笛草  796  95/09/25  光文社  カッパノベルス
人質カノン  1359  96/01/30  文藝春秋
本所深川ふしぎ草紙  427  95/09/01  新潮社  新潮文庫 み-22-5
龍は眠る  660  95/02/01  新潮社  新潮文庫 み-22-4
%
```

この場合は

```
% gawk '/9[56]\//' miyabe.new
```

とするのと同じです。両方ともデフォルトのアクション'{print;}' が使われていますが、それぞれ別のアクションを指定することもできます。

```
% gawk '/95\// {print $1;} /96\// {print $3;}' miyabe.new
96/09/01
96/10/10
96/10/30
96/07/15
```

```
とり残されて
鳩笛草
96/01/30
本所深川ふしぎ草紙
龍は眠る
%
```

95 年分は書名 ($) が、96 年分は発行日 ($3) が表示されています。スクリプトで書けば、次のようになります。

```
#!/usr/local/bin/gawk -f
# p_and_a.awk: ２つのパターンとアクションがある例

/95\// {
    print $1;
}

/96\// {
    print $3;
}
```

BEGIN や END はそれぞれ入力の開始前と終了後にマッチする特別なパターンです。それぞれ複数あっても構いません。すべて実行されます。ただし、アクションを省略することはできません。

3.4 AWK の変数と printf

複雑なプログラムを書くには変数が必要です[14]。変数名は C と同様、アルファベットで始まり、アルファベットと数字からなる文字列が使えます。正規表現で書けば [a-zA-Z][a-zA-Z0-9]*です。

ただ、C と違い、AWK には変数の型がありません。宣言する必要もありませんし、整数も浮動小数点数も文字列も気にする必要が (基本的に) なく、AWK が良きに取りはからってくれます。

書式を指定した AWK の出力には C とほぼ同じ形式の printf が使えます。最初のほうで見せた sin.awk の例をもう一度みましょう。

```
#!/usr/local/bin/gawk -f
# sin.awk:
BEGIN {
    for (i = 0; i <= 100; i++)
        printf("%1.6f\n", sin(i*3.1415/100));
}
```

変数 i を宣言なしで使っていますね。また、"%1.6f\n"は printf の書式指定で、もちろん f は浮動小数点数の意味です。詳細は C と全く同じなので説明はここでは省略しましょう。

ところで、このスクリプトのパターンは BEGIN しかありません。それ以外のパターンとアクションの組がありませんので、このスクリプトを実行すると、まずこの BEGIN のアクションを実行し、その後 (実行すべきものが何もないので) 何もせずに終了します。これは入力を受けとる必要がないときに良く使われる手法です。

[14] また怒られそうなことを書くし。って、誰に？

3.5 AWK の statements と組込関数

アクションの記述は式を並べることで行いますが、基本的に C と同じ式が書けます。また、次のような組み込みの数値関数が計算に使えます。表中の expr は数を値とする式です。

atan2(y,x)	y/x の逆正接 (ラジアン単位)
cos(expr)	余弦 (与える値はラジアン)
exp(expr)	指数関数
int(expr)	整数への変換
log(expr)	自然対数
rand()	0 から 1 の間の乱数
sin(expr)	正弦 (与える値はラジアン)
sqrt(expr)	平方根
srand(expr)	式 expr の値を乱数生成関数の種として用いる。式 expr が与えられなかった場合は、時刻が用いられる直前の種の値を返す。

さらに以下の文字列操作用の組込関数を持っています。

gsub(r,s,t)	文字列 t 中で正規表現 r にマッチした部分を s に置換する。置換した個数を返す。t を指定しなかた場合は$0 が用いられる。
index(s,t)	文字列 s に含まれる文字列 t の位置を返す。t が含まれていない場合は 0 を返す。
length(s)	文字列 s の長さを返す。s を指定しなかった場合には$0 の長さを返す。
match(s,r)	文字列 s で正規表現 r にマッチする位置を返す。マッチしない場合は 0 を返す。RSTART と RLENG の値が設定される。
split(s,a,r)	文字列 s を正規表現 r を用いて分割し、配列 a に格納する。r が省略された場合は FS が用いられる配列 a の内容は、いったんクリアされる。
sprintf(fmt,list)	フォーマット fmt に従って list を整形し、結果の文字列を返す。
sub(r,s,t)	gsub() と同様。ただし、最初にマッチした文字列のみが置換される。
substr(s,i,n)	文字列 s の i 文字目から n 文字の部分を返す。n が省略された場合、i 文字目以降の部分が返される
tolower(str)	文字列 str をコピーし、大文字をすべて小文字に変換したものを返す。アルファベットではない字は変化しない。
toupper(str)	文字列 str をコピーし、小文字をすべて大文字に変換したものを返す。アルファベットではない字は変化しない。

例えば gsub は次のように使います。

```
% gawk '{gsub(/新潮/, "しんちょう"); print; }' miyabe.csv
書名, 本体価格, 発行, 発行社, 文庫, 備考
火車,743,98/02/01, しんちょう社, しんちょう文庫 み-22-8,
かまいたち,505,96/09/01, しんちょう社, しんちょう文庫 み-22-6, 短編集
蒲生邸事件,1650,96/10/10, 毎日新聞社,
    (省略)
```

漢字の「新潮」が「しんちょう」に置き換わっています。これは単純な置き換えにすぎませんが、正規表現を少し覚えるともっと複雑なことができます。

少々実用的な例を上げましょう。あるディレクトリの中のファイル名が全部大文字のとき、それを小文字に変換したいとします。

```
% ls
AWK-INTRO.AUX  FS.0.AWK  MAKEFILE        SAMPLE1.DATA  SEC3.TEX
AWK-INTRO.DVI  FS.AWK    MAKEFILE.FINAL  SEC0.TEX      SIN.AWK
AWK-INTRO.LOG  GSUB.AWK  MIYABE.CSV      SEC1.TEX      SUM.AWK
AWK-INTRO.TEX  KUKU.AWK  MIYABE.DATA     SEC2.TEX
```

さて、ひとつひとつ

```
% mv AWK-INTRO.AUX awk-intro.aux
```

とやりますか？ 私ならこうします[15]。

```
% ls | gawk '{print "mv -v", $0, tolower($0);}'
mv -v AWK-INTRO.AUX awk-intro.aux
mv -v AWK-INTRO.DVI awk-intro.dvi
mv -v AWK-INTRO.LOG awk-intro.log
mv -v AWK-INTRO.TEX awk-intro.tex
mv -v FS.0.AWK fs.0.awk
    (省略)
```

この出力は、実行したい mv コマンドの羅列になっていますね。この出力をパイプで sh に渡してやれば、お望みの結果となります[16]。

```
% ls | gawk '{print "mv -v", $0, tolower($0);}' | sh
AWK-INTRO.AUX -> awk-intro.aux
AWK-INTRO.DVI -> awk-intro.dvi
AWK-INTRO.LOG -> awk-intro.log
AWK-INTRO.TEX -> awk-intro.tex
FS.0.AWK -> fs.0.awk
    (省略)
% ls
awk-intro.aux  fs.0.awk  makefile        sample1.data  sec3.tex
awk-intro.dvi  fs.awk    makefile.final  sec0.tex      sin.awk
awk-intro.log  gsub.awk  miyabe.csv      sec1.tex      sum.awk
awk-intro.tex  kuku.awk  miyabe.data     sec2.tex
%
```

このように AWK を使って実行したいコマンドの列を全部作らせておいて、それを sh に喰わせる（sh の入力にする）のです。

このアイデアに sub などの文字列操作関数などを組合わせると、沢山のファイルの名前の一部だけを小文字にするとか、ファイル名の一部に通し番号を振ったりすることが簡単にできます。

なお、mv にオプション-v を付けたのは、この実行例のように変更の様子を表示させたかっただけなので不用なら外してください。

[15] 本当は最近は ruby を使ってるんだけど。いや、こういう場合 tr だろ、って声も聞こえます。いろいろ解があるのが UNIX の楽しいところです。

[16] sh は、標準入力から入力された文字列をコマンドと解釈して順次実行します。どんな手段にせよ、実行したいコマンドの文字列を作って sh に渡すことさえできれば、そのコマンドを実行できます。

3.6 AWK の制御構造

AWK にも、if による条件分岐や while、for による繰り返しなどの制御構造があります。

```
if (condition) statement [ else statement ]
while (condition) statement
do statement while (condition)
for (expr1; expr2; expr3) statement
```

sin.awk の例を見れば分るように、条件式などの書き方も C と同じです。

```
#!/usr/local/bin/gawk -f
# sin.awk:
BEGIN {
    for (i = 0; i <= 100; i++)
        printf("%1.6f\n", sin(i*3.1415/100));
}
```

大小比較には<、>、<=、>=、==、!=、などが使えますし、&&、||、！　、も C の論理積、論理和、否定と同じです。

次のスクリプト nf.awk は、各レコード (各行) にちょうど 6 個のフィールドがあると期待しているとき、そのチェックを行うものです。

```
#!/usr/local/bin/gawk -f
# nf.awk: 各行のレコードに丁度 6 個のデータがあるか？
{
    if (NF > 6) {
        print NR "行目、データが多過ぎ (" NF "個:)", $0;
    } else if (NF < 6) {
        print NR "行目、データが不足 (" NF "個:)", $0;
    } else {
        print NR "行目、ＯＫ";  # この表示は余分かな。
    }
}
```

C と違うのは条件式のところに文字列と正規表現の比較が使えることです。次の例 reg.awk のようにチルダ ~ の左側に比較対象の文字列を、右側に正規表現を書きます。

```
#!/usr/local/bin/gawk -f
# reg.awk: 文字列と正規表現の比較
BEGIN {
    FS = ",";
}

{
    if ($0 ~ /96\//)            # 正規表現が文字列にマッチしたら真。
        print $1, $3;
}
```

これは次のスクリプト reg2.awk と同じ出力になります。

```
#!/usr/local/bin/gawk -f
# reg2.awk: 文字列と正規表現の比較 (2)
BEGIN {
    FS = ",";
}

/96\/ {
    print $1, $3;
}
```

また複数の式を { と } で囲ってブロックにして使うことも C と同じです。次の例は、九九を計算するスクリプト kuku.awk です。

```
#!/usr/local/bin/gawk -f
# kuku.awk: 九九の表を作る
BEGIN {
    for (i = 0; i < 10; i++) {
        for (j = 0; j < 10; j++)
            printf("%3d", i * j);
        printf("\n");
    }
}
```

実行結果はこんな具合です。

```
% ./kuku.awk
  0  0  0  0  0  0  0  0  0  0
  0  1  2  3  4  5  6  7  8  9
  0  2  4  6  8 10 12 14 16 18
  0  3  6  9 12 15 18 21 24 27
  0  4  8 12 16 20 24 28 32 36
  0  5 10 15 20 25 30 35 40 45
  0  6 12 18 24 30 36 42 48 54
  0  7 14 21 28 35 42 49 56 63
  0  8 16 24 32 40 48 56 64 72
  0  9 18 27 36 45 54 63 72 81
%
```

C より全然楽でしょう？

3.7 AWK の配列: 連想配列

AWK の配列には、C の配列にない便利な特徴があります。

3.7.1 数値データの処理

1 次元配列なら C と同様に次のように使えます。

```
#!/usr/local/bin/gawk -f
# array_1dim.awk: 0 から 9 までの和
BEGIN {
    for (i = 0; i < 10; i++)
        a[i] = i;
    sum = 0;
    for (i = 0; i < 10; i++)
        sum += a[i];
    print sum;
}
```

この array-1dim.awk は、わざわざ配列に保存する必要はありませんが、説明用ということで勘弁してください。

2 次元配列は C の書き方と少し違って、添え字をカンマ , で区切って指定します。行列の積を計算するスクリプト array-2dim.awk を書いてみました。

```
#!/usr/local/bin/gawk -f
# array_2dim.awk: 行列の積
BEGIN {
    # 行列 A
    a[0,0] = 1; a[0,1] = 2; a[0,2] = 3; a[0,3] = 4;
    a[1,0] = 1; a[1,1] = 2; a[1,2] = 3; a[1,3] = 4;
    a[2,0] = 1; a[2,1] = 1; a[2,2] = 3; a[2,3] = 4;
    a[3,0] = 1; a[3,1] = 1; a[3,2] = 1; a[3,3] = 4;
    # 行列 B
    b[0,0] = 0; b[0,1] = 0; b[0,2] = 0; b[0,3] = 1;
    b[1,0] = 0; b[1,1] = 0; b[1,2] = 1; b[1,3] = 0;
    b[2,0] = 0; b[2,1] = 1; b[2,2] = 0; b[2,3] = 0;
    b[3,0] = 1; b[3,1] = 0; b[3,2] = 0; b[3,3] = 0;
    # 積の計算: C = AB
    for (i = 0; i < 4; i++)
        for (j = 0; j < 4; j++) {
            c[i,j] = 0;
            for (k = 0; k < 4; k++)
                c[i,j] += a[i,k] * b[k,j];
        }
    # 以下は表示
    print "A=";
    for (i = 0; i < 4; i++) {
        for (j = 0; j < 4; j++)
            printf("%4d", a[i,j]);
```

```
            print "";               # 改行
        }
        print "B=";
        for (i = 0; i < 4; i++) {
            for (j = 0; j < 4; j++)
                printf("%4d", b[i,j]);
            print "";               # 改行
        }
        print "AB=";
        for (i = 0; i < 4; i++) {
            for (j = 0; j < 4; j++)
                printf("%4d", c[i,j]);
            print "";               # 改行
        }
    }
```

実行してみます。ちゃんと計算できてますね。

```
% ./array-2dim.awk
A=
   1   2   3   4
   1   2   3   4
   1   1   3   4
   1   1   1   4
B=
   0   0   0   1
   0   0   1   0
   0   1   0   0
   1   0   0   0
AB=
   4   3   2   1
   4   3   2   1
   4   3   1   1
   4   1   1   1
%
```

すこし応用を見てみましょう。

次の行列データをファイル matrix.data に保存しておき、スクリプト array-read.awk で読みこんでみましょう。

```
1 2 3
4 5 6
7 8 9
```

一行読むごとに row に+1 して行数を数えています。また NF がちょうど行列の桁数になります。

```
#!/usr/local/bin/gawk -f
# array_read.awk: 行列の読み込み
{
    row++;                      # 行数を数える
    column = NF;                # 列数を保存
    for (j = 1; j <= column; j++)
        a[row,j] = $j;
}

END {
    for (i = 1; i <= row; i++) {        # 表示
        for (j = 1; j <= column; j++)
            printf("a[%d,%d]=%d, ", i, j, a[i,j]);
        print "";
    }
}
```

実行してみましょう。

```
% ./array-read.awk matrix.data
a[1,1]=1, a[1,2]=2, a[1,3]=3,
a[2,1]=4, a[2,2]=5, a[2,3]=6,
a[3,1]=7, a[3,2]=8, a[3,3]=9,
%
```

この発展で、ファイルmatrix2.dataに保存された2つの行列の積を計算するスクリプトarray-read2.awkを作ってみます。

```
# # で始まる行はコメント
# ひとつめの行列
1 2 3
4 5 6
7 8 9

# ふたつめの行列
0 0 1
0 1 0
1 0 0
```

#から始まる行はコメント、また空行で 2 つの行列を分離しています。これを順に配列 a[1,i,j] と a[2,i,j] に読み込み、その積を c[i,j] に保存して次のような出力を得るようにしました。

```
% ./array-read2.awk matrix2.data
# 1 個めの行列:
# 1 2 3
# 4 5 6
```

```
# 7 8 9
# 2個めの行列:
# 0 0 1
# 0 1 0
# 1 0 0
# 積
3 2 1
6 5 4
9 8 7
%
```

ところで、array-read1.awk と違い、出力の形式をもとの入力の形式に近い形で出力させています。こうしておくと計算結果を再び array-read2.awk の入力にできます。以下、AWK の解説というよりコンピュータの使い方のコツのお話になります。

例えば今得た結果の 2 乗を計算してみましょう。まずシェル (csh や bash) の基礎知識の復習ですが、コマンドラインに複数のコマンドをセミコロン ; で区切って入力すると、それらのコマンドがその順番で実行されます。例えば、次のように入力すると command1、command2、command3 の順で実行されます。

```
% command1; command2; command3
```

そこで次のようにすれば、先の行列の積の計算を 2 回くりかえすことになります。

```
% ./array-read2.awk matrix2.data; echo; ./array-read2.awk matrix2.data
# 1個めの行列:
# 1 2 3
# 4 5 6
# 7 8 9
# 2個めの行列:
# 0 0 1
# 0 1 0
# 1 0 0
# 積
3 2 1
6 5 4
9 8 7

# 1個めの行列:
# 1 2 3
# 4 5 6
# 7 8 9
# 2個めの行列:
# 0 0 1
# 0 1 0
# 1 0 0
# 積
```

```
3 2 1
6 5 4
9 8 7
%
```

2つの行列の区切の空行を echo を使って入れています[17]。この出力結果は array-read2.awk の入力の書式に合っていますね。これをファイルに保存して再び array-read2.awk の引数にするか、もしくは次のようにパイプ | でスクリプトに渡してやればいいのです。

```
% (./array-read2.awk matrix2.data; echo; \
      ./array-read2.awk matrix2.data) | ./array-read2.awk
# 1 個めの行列:
# 3 2 1
# 6 5 4
# 9 8 7
# 2 個めの行列:
# 3 2 1
# 6 5 4
# 9 8 7
# 積
30 24 18
84 69 54
138 114 90
%
```

ここで注意しておくと、command1; command2; command3 と ; で区切ったコマンドの全ての出力を 1 つのファイル output に保存したり、別のコマンドにパイプで渡したいときはコマンド列全体を次のように ( ) で囲っておく必要があります。

```
% (command1; command2; command3) > output
% (command1; command2; command3) | other_command
```

また、この例のようにコマンドラインが長くなってしまうときは、コマンドラインの途中に \ を挟むと複数行に分けてコマンドを入力することができます。bash や tcsh、zsh であれば \ を挟まなくても、複数行に分けてコマンドを入力できます。試してみてください。

```
% (command1; command2; \
    command3) | other_command
```

実際のスクリプトは次の array-read2.awk です。

```
#!/usr/local/bin/gawk -f
# array_read2.awk: ２つの行列の読み込んで、積を計算する

BEGIN { num = 1; }              # num は、いくつ目の行列かを保存。
```

[17] echo を引数なしで実行すると空行が一つ出力できます。

```
NF == 0 {                          # 空行が行列の切れ目
    num = 2;
    row = 0;
}

(! /^\#/) && NF > 0 {              # コメントでなく、空行でないなら、、、
    row++;                         # 行数を数える
    column = NF;                   # 列数を保存
    for (j = 1; j <= column; j++)
        a[num, row, j] = $j;
}

END {
    for (k = 1; k <= 2; k++) {
        print "# " k "個めの行列:"
        for (i = 1; i <= row; i++) {    # 表示
            printf("# ");
            for (j = 1; j <= column; j++)
                printf("%d ", a[k, i,j]);
            print "";
        }
    }

    # row == column を仮定して積を計算
    for (i = 1; i <= row; i++)
        for (j = 1; j <= row; j++) {
            c[i,j] = 0;
            for (k = 1; k <= row; k++)
                c[i,j] += a[1,i,k] * a[2,k,j];
        }

    print "# 積"
    for (i = 1; i <= row; i++) {        # 表示
        for (j = 1; j <= column; j++)
            printf("%d ", c[i,j]);
        print "";
    }
}
```

ところで、このスクリプトでは、データファイルが正しい構造をしているかなどチェックしていません。チェックをプログラムに組み込むこともできますが、AWK はその場その場で即席にスクリプトを書いて実行するのが身上です。データファイルの書式のチェックが組みこまれた複雑なスクリプトを作るよりも、入力は正しいものとして処理するスクリプトを素早く作るほうが良いでしょう。入力のチェックが必要な

ら別にチェック用のスクリプトを書くほうが簡単ですし、信頼も置けます。

3.7.2 文字列を配列の添え字にする

AWK と C の配列で一番大きな違いは、配列の添え字に文字列が使えることです。これを**連想配列**と言い、テキスト処理がとても楽になります。

次のスクリプト array-str.awk は、宮部みゆきの文庫リスト miyabe.csv から出版社毎に何冊出ているかを調べます。

```
#!/usr/local/bin/gawk -f
# array_str.awk: 出版社毎の出版点数を調べる
BEGIN {
    FS = ",";
}

{
    publisher[$4] += 1;          # $4=出版社名を添え字に。
}

END {
    for (i in publisher)
        print i "は" publisher[i] "冊";
}
```

読み込んだフィールドのうち、出版社名 ($4) を配列 publisher の添え字に使っています。初めて使う変数は 0 に初期化されていますから、新潮社の本を読む度に publisher["新潮社"] の値に 1 を加えていけば、入力が終ったときには新潮社の出版点数が publisher["新潮社"] に保存されていることになります。

このスクリプトでは publisher に保存された結果をパターン END で出力しているのですが、連想配列の添え字には任意の文字列が使えますので、どんな添え字が使われているのかをプログラムの実行前に知ることはできません。そこで全要素を出力するために for の特別な書き方を使っています。

```
for (変数名 in 配列名) {
    処理;
}
```

このように記述すると、配列に使われている添え字全てに対して、添え字を変数に代入して「処理」を実行することを 1 回づつ行います。

実際に、このスクリプトを実行すると次のようになります。

```
% ./array-str.awk miyabe.csv
ＰＨＰ研究所は 1 冊
中央公論社は 2 冊
光文社は 5 冊
発行社は 1 冊
東京創元社は 2 冊
毎日新聞社は 1 冊
```

```
講談社は 2 冊
新人物往来社は 2 冊
朝日新聞社は 1 冊
大陸書房は 1 冊
新潮社は 10 冊
集英社は 1 冊
文藝春秋は 3 冊
%
```

ある文字列が配列の添え字に使われているかどうかを調べるには、if と in を組合せます。

```
if (調べたい文字列 in 配列名) {
    処理;
}
```

例えば、次のスクリプト array-str2.awk のようにします。

```
#!/usr/local/bin/gawk -f
# array_str2.awk: 新潮社からの出版物だけ表示する。
BEGIN {
    FS = ",";
}

{
    publisher[$4] += 1;          # $4=出版社名を添え字に。
}

END {
    name = "新潮社";
    if (name in publisher)
        print name "からは" publisher[name] "冊、出版されている。";
}
```

実行結果は次のようになります。

```
% ./array-str2.awk miyabe.csv
新潮社からは 10 冊、出版されている。
%
```

冊数でなく、書名を列挙したいときは次のスクリプト array-str3.awk のように、書名データをどんどん継ぎ足したものを出版社名を添字にした配列に保存しましょう。AWK では文字列を並べると自動的に連結された文字列が作られます。

```
#!/usr/local/bin/gawk -f
# array_str3.awk: 出版社毎の書名を表示する。
BEGIN {
    FS = ",";
```

```
}

{
    publisher[$4] = publisher[$4] "「" $1 "」"; # 書名$1 を「」で囲って連結
}

END {
    for (i in publisher)
        print i "からは" publisher[i] "が出版されている。";
}
```

これを実行すると、次のようになります。

```
% ./array-str3.awk miyabe.csv
```

ＰＨＰ研究所からは「初ものがたり」が出版されている。

中央公論社からは「今夜は眠れない」「夢にも思わない」が出版されている。

光文社からは「クロスファイア［上・下］」「スナーク狩り」「東京下町殺人暮色」「長い長い殺人」「鳩笛草」が出版されている。

発行社からは「書名」が出版されている。

（省略）

3.8 AWK のユーザ定義関数

スクリプトがちょっと大きくなってきたり、同じような処理を繰り返し書くことがあったら、AWK でも関数を定義して使うことができます。関数の定義は次の書式で書きます。

```
function 関数名 (引数の並び) {
    awk の命令
    awk の命令
    ...
    awk の命令
    return 戻り値
}
```

引数には数値でも文字列でも配列でもなんでも渡せます。return の戻り値も、数値と文字列が使えます。また再帰呼出しも出来ます。論より証拠、サンプル func.awk を見ましょう。

```
#!/usr/local/bin/gawk -f
# func.awk: 関数定義の例
BEGIN {
    a = "OK"; b = "OK"; c = "OK";
    print foo(1,2);
    print a, b, c;                      # a,b,c は変化しない
    print bar("AWK は", "便利だ");
    print a, b, c;                      # a,b,c は変化しない
    print "4! == " recursive(4);        # 4! == 24
```

```
}

function foo(a, b,      c) {            # c は局所変数のつもり
    c = a + b;
    return c;
}

function bar(a, b,      c) {
    c = a b;
    return c;
}

function recursive(a) {                 # 再帰呼び出しも可能
    if (a <= 1)
        return 1;
    else
        return a * recursive(a-1);
}
```

関数 foo は数値を引数 a と b で受けとって、その和を返す関数です。関数 bar は文字列を引数 a と b で受けとって、それを連結した文字列を返す関数です。

```
% ./func.awk
3
OK OK OK
AWK は便利だ
OK OK OK
4! == 24
%
```

どちらも仮引数に a、b、c を使っていますが、呼び出し側に同じ名前の変数があってもそれは変化しません。つまり、AWK の関数は C と同じで**値渡し**なのです。

ところで、呼び出しのところで foo も bar も引数 c に何も渡していません。これは AWK 流儀の**局所変数**の宣言です。実際には AWK の関数には局所変数がありません。AWK の関数の引数が値渡しであることを利用して、局所変数代わりにしているのです。

本当の引数 a,b に続いて局所変数用の引数 c を記述しますが、このサンプルのように、この 2 種類の変数を人間が区別しやすくするために間にスペースをたくさん置くことが慣習となっています。

この局所変数もどきがうまく動くのは、C と違って定義と呼出し時の引数の数が一致しているかどうかを AWK が確認しないからですが、これはある意味で**危険**なことも承知しておいたほうが良いと思います。

さて、配列を引数として渡すときは少し様子が違います。C と同じで、配列だけは**参照渡し**なので、関数内で配列要素を書き換えると呼び出し側でも値が変わります。

```
#!/usr/local/bin/gawk -f
# func2.awk: 関数に連想配列を渡す。
BEGIN {
```

```
    for (i = 0; i < 4; i++)
        a[i] = i;
    print "元の配列の内容を表示";
    for (i = 0; i < 4; i++)
        print i, a[i];
    foo(a);
    print "関数 foo 内で書換えると?";
    for (i = 0; i < 4; i++)
        print i, a[i];

    b["awk"] = "AWK"; b["perl"] = "PERL"; b["ruby"] = "RUBY";
    print "元の文字列";
    print b["awk"], b["perl"], b["ruby"];

    bar(b);
    print "関数 bar 内で細工すると";
    print b["awk"], b["perl"], b["ruby"];
}

function foo(a,         i) {
    for (i = 0; i < 4; i++)
        a[i] = -i;
}

function bar(a) {
    a["awk"] = reverse(a["awk"]);
    a["perl"] = reverse(a["perl"]);
    a["ruby"] = reverse(a["ruby"]);
}

# 文字列 str を逆転したものを返す ("abc" --> "cba")
# str 自体は変化しない。
function reverse(str,           a, i, j, t) {
    j = length(str);
    if (j <= 0)
        return;
    split(str, a, //);                  # 文字列を配列に分解
    for (i = 0; i < j/2; i++) {         # 配列の中身を逆順にする
        t = a[i]; a[i] = a[j-i]; a[j-i] = t;
    }
    t = "";
    for (i = 0; i < j; i++)
        t = t a[i];                     # 配列を文字列に戻す
    return t;
```

```
}
```

このスクリプト func2.awk を実行すると次のようになります。

```
% ./func2.awk
元の配列の内容を表示
0 0
1 1
2 2
3 3
関数 foo 内で書換えると?
0 0
1 -1
2 -2
3 -3
元の文字列
AWK PERL RUBY
関数 bar 内で細工すると
KWA LREP YBUR
%
```

4 AWK の一行野郎

わずか一行のプログラムで、AWK では現実的で便利な処理ができます。そんなプログラムを**一行野郎**とか one liners と呼びます。一行野郎は、スクリプトファイルを作らずに、次の例のようにコマンドラインにプログラムを直接書くのが粋です。ここに上げる例は、みな gawk のマニュアルに載っていたものです。例中の datafile は処理したいファイル名に置き換えてください。

```
% gawk '{ if (length($0) > max) max = length($0) } END { print max }' datafile
```

この一行野郎は、datafile の行の最大の長さを出力します。

```
% gawk '{if (length($0)>m) m=length($0)} END{print m}' datafile
```

一行野郎では変数名はうんと短かくて良いと思います。これは一つ目の一行野郎の変数 max を m に変えて、分りにくくならない程度に空白も取り除いたものです。

```
% gawk 'length($0) > 80' datafile'
```

80 文字を超える行だけを出力します。

```
% gawk 'NF > 0' datafile
```

少なくとも一つのフィールドがある行だけを出力します。つまり、空白文字だけの行を削除します。

```
% gawk 'BEGIN { for (i = 1; i <= 7; i++) print int(101 * rand()) }'
```

0 から 100 までの範囲で、ランダムな数を 7 個、出力します。

```
% gawk 'END { print NR }' datafile
```

レコードの値、つまり入力の行数を出力します。

```
% gawk 'NR % 2 == 0' datafile
```

入力の偶数行だけを出力します。これを使うと、C などで計算した結果をファイルに保存しておけば、刻み幅で計算結果を間引くことなどが簡単にできます。

```
% ls -lg FILES | awk '{ x += $5 } ; END { print "total bytes: " x }'
```

複数のファイルのサイズの総計を求めます。FILES にはファイル名を列挙したり、シェルのワイルドカードを使って対象ファイルを指定します。

```
% ls -lg FILES | awk '{ x += $5 } END { print "total K-bytes: " (x + 1023)/1024 }'
```

キロバイト単位にしたいのなら、最後で少し計算を加えれば良いですね。

5 AWK で簡易家計簿

一時期、一般家庭でのパソコンの利用のメインは、年賀状・暑中見舞の印刷と家計簿だと言われていたことがありました。今はどうなのでしょうか？

それはともかく、個人の家計簿をつけるのに、MS-Excel のような重装備の表計算ソフトや専用の家計簿ソフトを使う必要はないと思います。多機能でしょうけれど、使い方を覚えるのも大変だし、ソフトのバージョンが上ると昔のデータが読めなくなるとかいろいろ面倒の多いこともあるようです[18]。

基本的に家計簿に必要なことは日々の出費の総計が分ることと、せいぜい光熱費、食費などの項目毎の出費額ぐらいじゃないでしょうか？このように割りきってしまえば、AWK で簡単に処理できます。後で何か足りない機能があったら、少しづつ追加していけばいいのですし。[19]

ということで、ここでは応用例として簡易家計簿ソフトを作ってみましょう。最後には mule/emacs との連携も考えます。

5.1 第 0 版: 総和を計算するだけ

家計簿のデータは、AWK で処理するのですから、テキストファイルに保存します。つまり普通に使いなれたエディタでどんどん書いていけばいいことになります。AWK は行単位でデータを処理しますから、一項目一行という形式にしましょう。

一番安易には、最初に出費額を書いて空白で区切ってコメントを入れるのが楽でしょう。例えば次のようなデータファイル data.0 を作ります。各行の数字が金額で、空白で区切って適当にコメントを書きます。

```
2300        バス回数券
600         昼飯
1000        珈琲豆
3210        ガソリン代
800         夕飯
1050        傘
```

[18] OS のバージョンが上って動かなくなるとかね。莫迦みたいだと思います。
[19] プログラミングも楽しめるし。

次のようなスクリプト kakei0.awk も書きましょう。

```
#!/usr/local/bin/gawk -f
#kakei0.awk: 家計簿プログラム 第〇版

{
    total += $1;
}

END {
    print "出費の合計は " total " 円です";
}
```

そしてこのファイルに実行権限（第 3.2 節）を与えて、データファイルを引数にして実行すると出費の合計が出力されます。

```
% kakei0.awk data.0
出費の合計は 8960 円です
%
```

月単位でファイルを分けておき、毎日その日のデータを追加してはこのスクリプトを実行することで、その月の出費が分るわけです[20]。

あまりにもシンプルに思えるかもしれませんが、これが基本形です。あとはこれを少しづつ使いやすいものに変更していけば良いのです。

5.2 第 1 版: 日付を記録する

実は第 0 版では大切なことを無視しています。家計簿は日々の出費の記録ですから、いつの出費なのか、日付の記録が必要ですよね。そこで、次のように素朴に日付を記入したデータファイル data.0.date を作ってみます。

```
        11 月 25 日
2300            バス回数券
600             昼飯
1000            珈琲豆

        11 月 26 日
3210            ガソリン代
800             夕飯
1050            傘
```

これを第 0 版のスクリプトで処理すると。。。

```
% kakei0.awk data.0.date
出費の合計は 8982 円です
%
```

[20] 日単位でファイルにするという手もあります。それは貴方のお好みでどうぞ。

日付を追加しただけなのに合計がさっきと違っています。これは日付の月の 11 が加算されているのです。最初の数値の部分を金額だと AWK が勘違いしたのです。そこで、金額部分とコメント部分を明示的に分離しましょう (data.1)。ここでは分離記号にはセミコロン ; を使ってみます。

```
        ;11 月 25 日
2300    ; バス回数券
600     ; 昼飯
1000    ; 珈琲豆
        ;
        ;11 月 26 日
3210    ; ガソリン代
800     ; 夕飯
1050    ; 傘
```

そして、スクリプト kakei1.awk のほうも分離記号を ; に変更します。

```
#!/usr/local/bin/gawk -f
#kakei1.awk: 家計簿プログラム 第 1 版
#
# 項目の区切を ; にする。
# 家計簿のファイルの書式
#
#    金額 ; コメントを書ける

BEGIN {
    FS = ";";
}

{
    total += $1;
}

END {
    print "出費の合計は " total " 円です";
}
```

実行するとこんな具合です。

```
% kakei1.awk data.1
出費の合計は 8960 円です
%
```

5.3 第 2 版: 項目の順番を入れかえる

「ふつー、家計簿を書くとき、金額じゃなくて品名を先に書くんじゃない？」たとえば、こんな風に (data.2)。

```
11 月 25 日
バス回数券;       2300;              なんでも良いからコメントが書ける
昼飯;              600
珈琲豆;            1000

11 月 26 日
ガソリン代;       3210
夕飯;              800
傘;                1050
```

この書式変更に対応するためのスクリプト kakei2.awk の変更はわずかです。加算していくフィールドを変えるだけですね。

```
#!/usr/local/bin/gawk -f
#kakei2.awk: 家計簿プログラム 第 2 版
#
# コメントを先に書きたい？項目の区切を ; にする。
# 家計簿のファイルの書式
#
#     コメント ; 金額 ; さらにコメントを書ける

BEGIN {
    FS = ";";
}

{
    total += $2;
}

END {
    print "出費の合計は " total " 円です";
}
```

実行するとこんな具合です。

```
% kakei2.awk data.2
出費の合計は 8960 円です
%
```

5.4 第 3 版: 分類毎の合計も求める

食費や光熱費、交通費という分類を決めておいて、その分類毎の合計も求めることにしましょう。データファイルの書式は次のようにします。いままで唯のコメント欄だった第 3 フィールドに分類名を記入します。第 4 フィールド以降は自由にコメントが書けます (data.3)。

```
11 月 25 日
```

```
バス回数券;       2300;          交通費
昼飯;              600;           食費
珈琲豆;           1000;          嗜好品

11 月 26 日
ガソリン代;       3210;          交通費
夕飯;              800;           食費
傘;               1050;
```

スクリプト kakei3.awk のほうは、この第 3 フィールドの項目名そのものを連想配列の添え字に使ってしまえば、項目毎の総和は簡単に計算できますね。

```
#!/usr/local/bin/gawk -f
#kakei3.awk: 家計簿プログラム 第 3 版
#
# 分類毎の合計も欲しい。
# 家計簿のファイルの書式
#
#     コメント ; 金額 ; 分類 ; さらにコメントを書ける
#

BEGIN {
    FS = ";";
}

NF >= 2 {
    total += $2;
    if (length($3) > 0) {
        class[$3] += $2;
    } else {
        class["NONE"] += $2;     # 分類の指定なし
    }
}

END {
    print "出費の合計は " total " 円です";
    print "項目毎の出費は次の通り"
    for (i in class)
        print "\t" i "に関しては " class[i] " 円です";
}
```

実行するとこんな感じです。項目毎に合計されていますね。

```
% kakei3.awk data.3
出費の合計は 8960 円です
```

```
項目毎の出費は次の通り
    NONE に関しては 1050 円です
                交通費に関しては 5510 円です
                 食費     に関しては 800 円です
                 食費に関しては 600 円です
                嗜好品に関しては 1000 円です
%
```

でも、なんだか表示が汚いし、良く見ると食費が 2 つありますね。これはデータファイルを ; で区切っているので、項目名の前後にある余分の空白もフィールドの内容になっているからです。そこで、余分な空白を全部 処理前にとっぱらうことにしましょう (kakei4.awk)。それには gsub (第 3.5 節) を使います。

それともう一つ。項目名が書かれてない金額は、NONE を添え字とする連想配列の要素に加算しているので、それらの合計金額の表示での項目名が NONE になっています。これでは意味不明なので、もう少しましな表示に変更してみました。

```
#!/usr/local/bin/gawk -f
#kakei4.awk: 家計簿プログラム 第 4 版
#
# 家計簿のファイルの書式
#
#    コメント ; 金額 ; 分類 ; さらにコメントを書ける
#

BEGIN {
    FS = ";";
}

NF >= 2 {
    gsub(/[ \t]+/, "");
    total += $2;
    if (length($3) > 0) {
        class[$3] += $2;
    } else {
        class["NONE"] += $2;    # 分類の指定なし
    }
}

END {
    print "出費の合計は " total " 円です";
    print "項目毎の出費は次の通り"
    for (i in class)
        if (i == "NONE")
            print "\t 分類のない項目が " class[i] " 円です";
        else
            print "\t" i "に関しては " class[i] " 円です";
```

```
}
```

実行結果は次の通りです。

```
% kakei4.awk data.3
出費の合計は 8960 円です
項目毎の出費は次の通り
    分類のない項目が 1050 円です
    食費に関しては 1400 円です
    交通費に関しては 5510 円です
    嗜好品に関しては 1000 円です
%
```

5.5 第 5 版: その日の出費は？

その月の出費の合計のほか、当日の出費が知りたいでしょうか? でしたら、当日の出費を計算する変数を追加して、日付の指定が第 1 フィールドに表われるたびにそれを 0 にしてやれば良いでしょう (kakei5.awk)。

```
#!/usr/local/bin/gawk -f
#kakei5.awk: 家計簿プログラム 第 5 版: 最後の日の集計も表示。
#
# 家計簿のファイルの書式
#
#    コメント ; 金額 ; 分類 ; さらにコメントを書ける
#

BEGIN {
    FS = ";";
}

/^\s*[0-9]+月 [0-9]+日/ {        # 行頭から日付が書かれていたら、
    dtotal = 0;                  # 日付単位の集計結果をクリア。
    for (i in dclass)
        dclass[i] = 0;
}

NF >= 2 {
    gsub(/[ \t]+/, "");
    total += $2;
    dtotal += $2;
    if (length($3) > 0) {
        class[$3] += $2;
        dclass[$3] += $2;
    } else {
        class["NONE"] += $2;     # 分類の指定なし
```

```
        dclass["NONE"] += $2;   # 分類の指定なし
    }
}

END {
    print "出費の合計は " total " 円です";
    print "項目毎の出費は次の通り"
    for (i in class)
        if (i == "NONE")
            print "\t 分類のない項目が " class[i] " 円です";
        else
            print "\t" i "に関しては " class[i] " 円です";

    print "";

    print "今日の出費は " dtotal " 円です";
    print "項目毎の出費は次の通り"
    for (i in dclass)
        if (i == "NONE")
            print "\t 分類のない項目が " dclass[i] " 円です";
        else
            print "\t" i "に関しては " dclass[i] " 円です";
}
```

実行結果は次の通りです。

```
% kakei5.awk datafile
出費の合計は 8960 円です
項目毎の出費は次の通り
    分類のない項目が 1050 円です
    食費に関しては 1400 円です
    交通費に関しては 5510 円です
    嗜好品に関しては 1000 円です

今日の出費は 5060 円です
項目毎の出費は次の通り
    分類のない項目が 1050 円です
    食費に関しては 800 円です
    交通費に関しては 3210 円です
    嗜好品に関しては 0 円です
%
```

出力部分を関数にしてもよかったかな。

5.6 Mule/Emacs との連携

もし使っているエディタが mule/emacs なら、次のようなもの (kakei.el) を~/.emacs に書いておくと、M-x kakei とタイプすれば編集中のファイルを家計簿のデータとして処理してくれます。

```
;;
;; 家計簿スクリプトを呼ぶ emacs-lisp プログラム
;;

(defvar kakei-script "kakei.awk" "家計簿を計算するプログラム")
(defun kakei ()
  (interactive)
  (shell-command-on-region (point-min) (point-max) kakei-script))
```

"kakei.awk"は実際に利用するスクリプトのファイル名に変更しておいてください。スクリプトには実行権限 (第 3.2 節) を与えて、実行パスの通ったディレクトリ (例えば~/bin) に置いておくことが前提です。

6 おしまいに

コンピュータを使うときには定型の仕事はコンピュータにまかせようという心掛けをして、カーニハンとプローガーの名著「ソフトウェア作法」の次の言葉を心にとめておくことは、コンピュータに人間が使われないために有効だと思います。

> やりたい仕事の 90%をコンピュータがこなしてくれれば、(そのプログラムの) ユーザは満足する

100%の完璧を求めることはない、ほとんどの作業を正しく処理できるプログラムが書ければよい、最後の仕上げは人がやればいいんだよ、ということなのです[21]。ただし、仕上げに人手をかける分だけ、プログラミングにかかる手間は減らしておかないとなんのためのコンピュータ利用かわからなくなります。そして、AWK を始めとするスクリプト言語はプログラミングの手間の少ない言語になっています。

AWK の参考書としては次の 4 つが良いでしょう。

1. 「たのしい UNIX」 坂本文著 (アスキー出版局)
2. 「AWK を 256 倍使うための本」志村拓、鷲北賢、西村克亘 (アスキー出版局)
3. 「プログラミング言語 AWK」 エイホ、ワインバーガー、カーニハン ()
4. GNU AWK マニュアル

「プログラミング言語 AWK」は AWK の設計者自らの解説本ですが、著者は読んでません。最後の「GNU AWK マニュアル」は gawk のマニュアルですが、入門書としても素晴しい出来だと思います。web で検索すると日本語訳が見付かります。

また、近頃はやりのスクリプト言語と言えば、perl や ruby でしょう。どちらも AWK とは較べものにならないほど多機能です。AWK にできて perl や ruby にできないことはなく、AWK の次にスクリプト言語を触ってみたいなら perl や ruby がお勧めです。どちらも入門書がたくさん売られていますし[22]、特に ruby は日本人が作っている言語ですから web にも日本語の解説ページが豊富にあります。

[21] もちろん、これは仕事内容によります。たとえば組版ソフトの場合、コンピュータの出力が最終出力になっていないと無意味でしょうし。

[22] ruby の入門書がたくさん出始めたのは、最近。ありがたいことです。