初心者用
Perl 講座

第 3 版 2008 年 4 月 9 日

文責：斎藤輪太郎・永安悟史

0. はじめに

Perl はテキスト処理言語です。与えられたテキストファイル(cat や more で中味を見ることができるファイル)を加工して出力する時に威力を発揮します。awk もテキスト処理が得意ですが、機能が豊富で使い方次第で awk より大きな仕事ができます。このテキストではいくつかの例題をこなしながら Perl をある程度使えるようになることを目指します。

1. 文字列の表示

まずは、なにか文字列を出力してみるところから始めましょう。以下のようなファイルを作成して下さい。ファイル名を hello.pl とします。

```
#!/usr/bin/perl

print “Hello, world!¥n”;
```

上記の Perl のプログラムを作成したら、以下のようにして実行権を与えましょう。

```
chmod +x hello.pl
```

そして以下のようにして hello.pl を実行します。

```
./hello.pl
```

Hello, world!という文字列が出力されましたか？
このように print は次に続く文字列を出力する機能があります。次に print の行を、

```
print 1+2;
```

に変えて実行してみましょう。どうなりましたか？print に数式が続く場合、その数式が計算されて答えが出力されます。

課題 1-1：123 x 456 を計算させてみましょう。掛け算には * を使います。

2. ファイルのオープンと行単位の読み込み

以下のようなファイルを作ってみましょう。ファイル名を food.txt とします。

```
I like an apple.
He ate a banana.
I cooked some corn.
She has some donuts.
```

次に以下のような Perl スクリプトを書いてみましょう。ファイル名を eat.pl とします。

```
#!/usr/bin/perl

open(FILE, “food.txt”);
while(<FILE>){

   print $_;

}
close FILE;
```

※ #!/usr/bin/perl の行はシステムによって変える必要があります。which perl で調べて下さい。

Perlではセミコロン ； がしばしば文を区切る上で重要な役割を果たします。忘れずに付けましょう。

上記の Perl のスクリプトを作成したら、以下のようにして実行権を与えましょう。

```
chmod +x eat.pl
```

そして以下のようにして Perl スクリプトを起動します。

```
./eat.pl
```

food.txt の内容がそのまま出力されましたか？

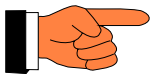
ではスクリプトの説明をしましょう。

- #!/usr/bin/perl は以下のスクリプトが Perl スクリプトであることを指定しています。Perl スクリプトには必ずつけましょう。
- open(FILE, “food.txt”); は”food.txt”というファイルを扱う（オープンする）ことを宣言します。そして FILE という変数（ファイルハンドル）でこのファイルを管理します。ファイルの読み書きは全てこの FILE というファイルハンドルを使って行います。FILE でなくても、HANDLE など任意の名前をつけることができます。
- while(<FILE>){　処理　}で FILE(この場合、food.txt)が一行ずつ読み込まれ、一行読み込まれるたびに{　　}の中の処理が行われます。読み込みはファイルが終わるまで行

われます。つまり通常は、food.txt の行数分だけ{　　}の中が実行されます。

- print $_で読み込んだ一行を表示します。$_には読み込んだ一行が一時的に格納されています。上述の while 文の中にあるので、結局全ての行が表示されることになります。
- close FILE でファイルを閉じます。

課題 2-1：2 つのファイル（例えば、food.txt とてきとうに作った food2.txt）を続けて出力するスクリプトを書きましょう。



open(FILE, $ARGV[0]); として ./eat.pl file1 して実行すると file1 の内容が表示されます。$ARGV[0]はプログラムを呼び出すときの 1 番目の文字列を、$ARGV[1]は 2 番目の文字列を表します。

3. 簡単な変数の使い方

変数は数値や文字列を一時的に入れておく入れ物のような存在です。C 言語のようにあらかじめ宣言する必要はありません。変数の先頭には $ がつきます。

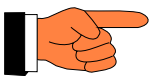
```
$val = 150;                  # 1
$name = “Watson Crick”;      # 2
```

#1 は 150 を$val という変数に代入します。
#2 は”Watson Crick”という文字列を$name に代入します。

変数で演算を行うことも可能です。

```
$val = 10 + 20;              #3
$val = $val1 + $val2;        #4
$val = $val + 1;             #5
```

#3 は 30 を$val に代入します。
#4 は変数$val1 と$val2 を足した結果を$val に代入します。
#5 は$val に 1 を加え、それを新しい$val の値とします。



”+” は基本的には数値計算に使います。文字列をつなげるにはピリオド”.”を使います。 例：$str1 = “abc”;$str2 = “def”;$str3 = $str1 . $str2;

課題 3-1： 変数$var1 に 123、$var2 に 456 を代入し、この二つを足した数を$var3 に代入

してみましょう。また答えを print $var3 で表示しましょう。
課題 3-2：ファイル中の行数を数える Perl スクリプトを書きましょう。

4. 基本的な条件分岐：if 文

ある条件によって処理の内容を変えたいということはよくありますね。多くの他の言語と同様、Perl でも if 文を使って条件分岐を行います。以下のプログラムを打ち込んで動作を確認しましょう。

```
#!/usr/bin/perl

open(FILE, “food.txt”);
while(<FILE>){

   if(length($_) > 15){ print $_;}

}
close FILE;
```

・ファイルを一行ずつ読むのは前と一緒です。
・length は文字列の長さを求めます。length($_)は読み込んだ行の長さを意味します。
・if(length($_) > 15){ print $_; }はもし読み込んだ行の長さが 15 を超えるなら、その行を表示せよという意味です。
結果的に、一行の長さが 15 を超える行だけが表示されます。

---

if の構文は以下の通りです。

if(条件){ 条件にあったときの処理 }
elsif(2 番目の条件){ 条件にあったときの処理 }
else { どの条件にも合わなかったときの処理 }

if(　　)の後の処理は必ず｛ と ｝で囲うようにしましょう。

課題 4-1：ファイルの中味を表示するスクリプトを書きましょう。但し、一行の長さが 15 文字を超える行に対しては行頭に”!!”を,5 字以下の行に対しては行頭に”!.”を、それ以外にたいしては行頭に..を付加します。

数値の比較には、>, >=, ==, <=, <　などが使われますが、文字列の比較には eq(等しい)、ne(等しくない)などが使われます。

5. while 文によるループ

これまでのプログラムでは、実行の流れがかならず上から下でした。ここでは、プログラムの流れを下から上に戻す手法（ループ）を学びます。一番簡単な while 文から学びましょう。
while 文はある条件を満たしている間は一定範囲のスクリプトを実行し続ける文で、以下のような構造をしています。

while(条件){
　　処理
　　:
　　:
}

条件を満たしている間、{　}内の処理が繰り返されます。以下のプログラムを入力して実行してみましょう。

```
#!/usr/bin/perl

$num = 0;
while($num <= 5 ){
    print “$num “;
    $num ++;
}
```

実行すると以下のような結果が得られるはずです。

```
0 1 2 3 4 5
```

・$num = 0 で最初に$num を 0 にセットします。
・$num が 5 以下の間、print $num(数の出力)と$num ++($num の値を 1 つ増やす)が繰り返されます。
・結果的に$num が 0 から 5 まで増え、それが出力されます。

課題 5-1：while 文を使って 2 x 3 x 4 x 5 x 6 x 7 x 8 x 9 を計算しましょう。

6. for 文によるループ

for 文も while 文と同じようなループを制御する文です。以下のような構造をしています。

```
for(初期設定;継続条件;更新){
        処理
          :
          :
}
```

・for に入るとまず初期設定が実行されます。
・その後、{ }の中の処理が繰り返されますが、その前に毎回継続条件が確かめられ、条件に合わなくなると、for 文を抜け出します。
・for 文の中の処理が終わると毎回更新が実行されます。

先ほど while 文の例として作ったスクリプトは for 文では次のように書きます。

```
#!/usr/bin/perl

for($num = 0;$num <= 5;$num ++){
    print “$num “;
}
```

課題 6-1：for 文を使って 2 x 3 x 4 x 5 x 6 x 7 x 8 x 9 を計算しましょう。

while 文や for 文のループを強制的に抜け出す命令として、last、ループの次のサイクルを強制的に開始する命令として next などがあります。

7. 正規表現

Perl では正規表現と呼ばれる表現法を使うことができます。
正規表現は簡単にかみくだいて言えば、１つの表現で複数のパターンを指定できる表現法です（この説明は情報科学的に正確とはいえませんが)。
例えば正規表現で t..e は tree,take,tape など複数のパターンを表します。正規表現を使うことによってパターンの検索に融通が利くようになるわけです。ピリオド”.”は任意の一文字を表します。従って..で任意の二文字を表しています。

正規表現の例をいくつか挙げておきます。

^	行頭
.	任意の１文字
[c1c2..cn]	c1～cn の中の任意の文字
[^c1c2..cn]	c1～cn の中にない任意の文字
r*	r に適合する文字列の 0 個以上の連続
r+	r に適合する文字列の 1 個以上の連続
¥.	ピリオド(¥で正規表現の特別な意味をなくす)
was|were|will	was,were,will のいずれか

^The は行頭の The にマッチします。
[acgt]は a,c,g,t の任意の一文字にマッチします。
[acgt]*は a,c,g,t から構成される任意の長さの文字列（長さ 0 も含む）にマッチします。
[a-z]はアルファベットの小文字にマッチします。
[^A-Za-z]はアルファベット以外の文字にマッチします。

よくある使い方が、「ある正規表現がある文字列にマッチするか？」というところでしょう。
Perl ではそれを =~ / 正規表現 / で表します。

```
while(<FILE>){

```

```
    if($_ =~ /[tT]he/){ print $_; }

}
```

ファイルを読み込み、the および The が含まれている行を表示します。

課題 7-1：行頭にtheおよびTheがある行を表示しましょう。

8. 文字列の置換

Perl では文字列中に含まれるあるパターンを別の文字列に置き換えることがとても簡単にできます。以下の構文を使います。

変数 =~ s/式 1/式 2/g;

変数（文字列が入っている）の中に入っている式 1 のパターンを全て式 2 に置換します。結果は変数自体に格納されます。

```
#!/usr/bin/perl

$seq = “aaaatttt”;

$seq =~ s/a/T/g;
print “$seq¥n”;
$seq =~ s/t/A/g;
print “$seq¥n”;
```

以下のような実行結果が得られるはずです。

```
TTTTtttt
TTTTAAAA
```

課題 8-1：テキストファイル中の”the”という単語を”a”に全て変換するようなスクリプトを書きましょう。

9. 配列変数

似たようなデータの集合を同じ変数名で一元的に管理したいときに配列はとても便利です。

例えば、

```
@name = ( “Thomas”, “John”, “Angela”, “Joanna”);
```

とすれば、$name[0]は”Thomas”,$name[1]は”John”を表します。
この例で分かるように使い方は、初期設定は、

@変数名 =(要素 0,要素 1,要素 2, …)

で行い、その後配列の要素を参照・変更するには

変数名[添字]

とします。例えば$var1[10]は var1 という配列の 10 番目(0 から数えるなら 11 番目)の要素を表します。
※今まで使ってきたような配列変数でない変数のことをスカラ変数と呼びます。
次のスクリプトをみてみましょう。

```
#!/usr/bin/perl

@name = ( “Thomas”, “John”, “Angela”, “Joanna”);
for($i = 0;$i <= 3;$i ++){
   print “$name[$i] “;
}
print “¥n”;
```

以下のような出力が得られるはずです。

```
Thomas John Angela Joanna
```

上のスクリプトは foreach 文を使って以下のように書き換えても同じ結果が得られます。

```
#!/usr/bin/perl

@name = (“Thomas”, “John”, “Angela”, “Joanna”);
foreach $person (@name){
   print “$person “;
}
print “¥n”;
```

foreach 文は次のような構造をしています。

foreach 変数 (配列) {

処理

}

配列の中の要素が先頭から順番に変数に代入されます。そして代入されるたびに処理が行われます。

課題 9-1：テキストファイルを行末から行頭へ向かって表示するようなスクリプトを書きましょう。

出力例：

People in some countries say that Japanese people work very much. However, the way they work is not very efficient. If they work in more efficient way, Japan may develop more.	→	develop more. more efficient way, Japan may not very efficient. If they work in However, the way they work is Japanese people work very much. People in some countries say that

10. 連想配列

連想配列（ハッシュ）という配列があります。これはキーとそれに対応する値を結びつけるものです。例えば、今%furuta という連想配列を考えます。

```
$furuta{ “Height” } = “180cm”;
$furuta{“weight”} = “80kg”;
$furuta{“age”} = “35”;
```

とすると、$furuta{ “Height” }に対応する値は 180cm になります。

```
print $furuta{“Height”};
```

としてみましょう。

また

```
%furuta = (“Height”  => “180cm”,
           “weight” => “80kg”,
           “age”    => “35”);
```

としておいて

```
print $furuta{“Height”};
```

としても同じ答えが得られます。

このように連想配列に値を代入するときは、

%連想配列名 =(“キー1” => “値 1”,”キー2” => “値 2”,”キー3” => “値 3”,. . . .);

とするか、

$連想配列名{“キー”}= 値;

とします。

課題 10-1：連想配列を使って何人かの友達の名前とログイン名を対応させてみましょう。そしていくつか print 文を使って表示させてみましょう。

11. 関数

関数は特定の処理を行うための機能単位です。例えば、以下の文字列を出力する処理を考えてみましょう。

```
print “****************************************¥n”;
print “ This work was done by Yukichi Fukuzawa¥n”;
print “****************************************¥n”;
print “ Last update: 2003/10/5¥n”;
```

上記のような処理がプログラムの至るところで必要なとき、上記４行を至るところで書くのは大変です。そこで例えばプログラムの先頭の方で、

```
sub print_author {
	print “****************************************¥n”;
	print “ This work was done by Yukichi Fukuzawa¥n”;
	print “****************************************¥n”;
	print “ Last update: 2003/10/5¥n”;
}
```

としておくと、上記処理が必要な箇所で、

```
&print_author
```

とするだけで処理が実行されます。４行が１行に圧縮されたことになります。なお、Yukichi Fukuzawa のところをプログラムのいろいろな箇所で変えたい場合、

```
sub print_author {
	my $author_name = $_[0];
	print “****************************************¥n”;
	print “ This work was done by “, $author_name, “¥n”;
	print “****************************************¥n”;
	print “ Last update: 2003/10/5¥n”;
}
```

とすれば、print_author(“Yukichi Fukuzawa”);と呼び出すだけで、同じ処理が実行されます。ここで、print_author(“Shigenobu Okuma”)とすれば、Yukichi Fukuzawa が Shigenobu Okuma に変わって実行されます。なおこの場合の括弧の中の”Yukichi Fukuzawa”や、”Shigenobu Okuma”のことを引数と呼びます。

独自の関数を作ることは、プログラミング作業の効率化に大きな役割を果たします。Perl の関数の概念は、他のプログラミング言語と大差ありませんので、ここでは細かな言語的

な部分を解説します。

11.1 関数の呼び出し

Perl では、関数を呼び出す場合には、関数名の最初に「&」の記号を付加して呼び出します(引数がない場合、&は省略可)。例えば、「func1」という関数を呼び出す場合には、

```
&func1;
```

のように記述します。引き数や戻り値（後述）を利用する場合には、

```
$result = &func1($param);
```

のように記述します。この場合、$param が引き数になり、$result が戻り値になります。

11.2 関数の定義

関数は、サブルーチン（sub-routine）である、という観点から、

```
sub function {
   ....
   ....
   ....
}
```

のように、関数名の前に「sub」を付加し、「{、}」のブロックで囲って記述します。

11.3 引き数と戻り値

関数では、その関数に値を与えて（これを「引き数」という)、その結果（「戻り値」という）を受け取ることができます。引き数と戻り値を効果的に利用することによって、プログラミングの効率を飛躍的に向上させることができます。

関数の種類としては、以下のような組み合わせがありえます。

① 引数も戻り値も無い場合
② 引数があり、戻り値が無い場合
③ 引数が無く、戻り値がある場合
④ 引数も戻り値もある場合

① 引き数も戻り値も無い場合

「引き数も戻り値も無い場合」というのは、プログラムの処理状況に関わらず、決まりきった処理を行う場合に主に用いられます。

例えば、そのプログラムの使い方を表示するような関数を利用する場合、単に文字列だけを表示すればいい訳ですから、値を渡す必要も、渡される必要もありません。そのような場合、

```
sub print_usage {
  print "Usage: progname [option...] [filename]¥n";
  print "¥n";
  print "      -h:    Print this help¥n";
  print "¥n";
}
```

のように関数を記述することによって、

```
&print_usage;
```

という関数呼び出しで、ヘルプの表示を行うことができるようになります。

② 引き数があり、戻り値が無い場合

引き数がある場合、引き数は、関数の内側で「@_」という変数で受け取ります。

「print_number」という名前で与えた引き数を表示する関数を定義する場合には、

```
&print_number(4);

sub print_number {
    $num = $_[0];              # @_ の一つ目の要素だけを取り出す

    print $num."¥n";
}
```

とすることで、値を受け取ることができます。

また、引き数は一つとは限りません。複数渡す場合には、

```
&print_numbers(10,20,100);

sub print_numbers {
    ($num1,$num2,$num3) = @_;     # @_ の 1,2,3 つ目の要素を取り出す
}
```

とすることで、複数の引き数を渡すことができます。

③ 引き数が無く、戻り値がある場合
④ 引き数も戻り値もある場合
については各自考えましょう。

ある二つの数を足して、その結果を得るための関数を定義してみます。

```
sub calc_sum {
  ( $num1,$num2 ) = @_;

  $num3 = $num1 + $num2;

  return $num3;
}
```

関数の返す値は、「return」文で指定します。このようにすることによって、

```
$result = &calc_sum(3,6);

print $result;
```

と関数の実行結果を得ることができます。

11.4 引き数、戻り値にできる変数の種類

Perl では、さまざまな種類の変数をパラメータにできます。

① スカラ変数（$foo, $bar などで記述される変数）
② 配列変数（@foo のように記述される変数）
③ 連想配列（ハッシュ。%foo のように記述される変数）
④ その他いろいろ（ファイルハンドルとか）

スカラ変数を複数指定したり、配列変数や連想配列をまとめて渡したり受け取ったりすることができます。

11.5 局所変数

Perl では、基本的に変数を宣言する必要はなく、記述した時点で変数が作成されます。このことは、ごく小さいプログラムを書いている場合には有効ですが、一画面に収まらないようなプログラムや、関数を利用するようなプログラムでは破綻します。

Perl では、基本的に変数のすべてがグローバル変数として扱われるため、プログラムの規模の拡大と同時に変数のスコープ（有効範囲）に関する問題が生じます。その解決の為に、ブロック（主に関数）内でのみ有効なローカル変数を利用することができます。

```
sub function {
    my($localval);
}
```

のように、「my」を冠することによって、その変数はそのブロックにローカルなスコープを持つようになります。

グローバル変数を多用するプログラム（特に関数）は、カットアンドペーストでプログラムの再利用を行う場合に支障をきたします。引き数や戻り値以外に、動作に影響する要素があるためです。それが多くなると、プログラムの再利用は破綻します。書き捨てのプログラムでない場合には、基本的に関数内でグローバル変数にアクセス（読み取り、変更など）をするのは、極力避けるようにしましょう。

例を以下に示します。これは配列変数の全要素を表示するものです。

```
&print_array(@a);

sub print_array {
    my(@array) = @_;

    foreach (@array) {
        print $_."¥n";
    }
}
```

課題 11-1：2 つの整数値を受け取り、その数値と間の整数をすべて足し合わせて結果を返す関数 total を定義しましょう。引き数は正の整数であるとし、最初の引数が 2 番目の引数より常に小さいと仮定します。例えば total(3,6)は 18 となります。

12. 組み込み関数

Perl にはあらかじめ組み込まれている関数がいくつかありますのでいくつかを紹介します。

 これらの関数を使うときには、&などをつける必要はありません。

length($str1) … 文字列$str1 に含まれる文字数を返します。
使用例：

```
print length(“abcde”);
```

substr($str1, $offset, $len) … $str1 の$offset の位置から$len 文字分の文字列を返します。
使用例：

```
print substr(“abcdefghijklmn”, 5, 3);
```

index($str1, $str2) … 文字列$str1 の中で文字列$str2 が見つかった位置を返します。
使用例:

```
print index(“abcdefghijklmn”, “ghi”);
```

split(/delimit/, $str1) … 文字列を delimit を境に配列に分解します。
使用例:

```
$line = “a-b-c-d-e”;
@letter = split(/-/, $line);
print “$letter[0] $letter[2]¥n”;
```

printf(….) … C 言語などで使われるものとほぼ同じです。ひじょうに多機能で詳しいことは省略します。
使用例:

```
printf(“%s --- %dC¥n”, “Temperature”, 20);
```

課題 12-1：実際に使用例を打ち込んでどのような出力結果が得られるか、確かめましょう。
課題 12-2：文字列を逆にする関数 original_reverse を length や substr を使って自分で作ってみましょう。

13. 様々なファイル操作

まずファイルへの書きこみを行う方法を学びましょう。

まずファイルを書きこみ用にオープンします。>を付けると書きこみ用にオープンするという意味になります。

```
open(W_FILE, “> outfile”);
```

これで outfile という名前のファイルが書きこみ用にオープンされました。
そして print でファイルに書きこむことができます。

```
print W_FILE “Hello¥n”;
```

W_FILE など、ファイルハンドルの名前はもちろん、任意の名前にすることができます。
書きこむ必要がなくなったら、

```
close W_FILE;
```

でファイルを閉じます。これで cat コマンドを使って outfile の中身を見ると、Hello と書きこまれていることが分かります。

課題 13-1: 自分の名前を myname というファイルに print を使って書きこんでみましょう。

すでに学んだように、ファイルを読みこむとき、

```
while(<FILE>){
    :
}
```

とすることにより、上から下まで一行ずつ読まれていきます。しかし、seek を使うことにより、任意の位置から読み取りを開始することができます。

```
seek(FILE, 0, 0);
```

とするとファイルの最初から読みなおすことができます。また、

```
seek(FILE, 100, 0);
```

とすると、ファイルの先頭から数えて 100 文字目（0 文字目から数え始める。また改行などを含む）からファイルを読みこむことができます。

seek 関数のの最初の文字はファイルハンドル名を、２番目の数値は何文字目かを、３番目の数値 0 はファイルの先頭からという意味を表します。以下のプログラムを catfrom.pl というファイル名で作成してみましょう。

```
#!/usr/bin/perl

```

```
open(FILE, $ARGV[0]);
seek(FILE, $ARGV[1], 0);

while(<FILE>){
   print $_;
}
close FILE;
```

そして textfile という 10 文字以上のファイルを作成し、

```
./catfrom.pl textfile 10
```

とすると、10 文字目からファイルの内容が表示されます。

なお、<FILE>はファイルから一行を読みとりますが、read 関数を使うと、任意の文字数を指定した変数に格納することができます。以下のプログラムを打ち込んで動作を確認してみましょう。

```
#!/usr/bin/perl

open(FILE, $ARGV[0]);

seek(FILE, $ARGV[1], 0);
read(FILE, $string, 20);
print $string;

close FILE;
```

上のプログラムで、read(FILE, $string, 20)はファイル FILE から 20 文字読み込んで、それを$string に格納します。

課題 13-2：指定したファイルの指定された場所から指定された文字数を表示するプログラム cat_from_to.pl を作成しましょう。例えば、

```
./cat_from_to.pl textfile 50 20
```

とすると、textfile の 50 文字目の文字から 20 文字を表示するようにしましょう（改行も一文字と数えて構いません）。

14. コマンドへの入出力

UNIX では ls（ファイルの一覧）、pwd（現在のディレクトリの表示）など、様々なコマン

ドがあります。ここでは、そのコマンドの出力をプログラムで受け取る方法を説明します。

```
open(FILE, "UNIX コマンド |");
```

このようにすると、コマンドの出力をあたかもファイルからの出力であるかのように受け取ることができます。例えば、

```
open(FILE, "ls -l |");
```

とすると、ls -l のコマンドの出力を一行ずつ受け取ることができます。
そして、

```
open(FILE, "ls -l |");
while(<FILE>){

   print $_;

}
```

のようにすると、ls -l の出力が一行ずつ$_に入り、それがそのまま出力されます。その途中で出力を加工することもできます。

```
open(FILE, "ls -l |");
$dummy = <FILE>;
while(<FILE>){
   chop;
   $_ =~ s/ +/ /g;
   @line = split(/ /,$_);
   print “You have file “, $line[8], “ in this directory.¥n”;

}
close FILE;
```

ls -l では最初の一行目の出力が合計ファイル数の情報で、ファイル名の情報ではないので、$dummy = <FILE>;で最初の一行の出力を$dummy にとりあえず格納します。chop は$_ の最後の改行を取る関数です（この行を除いたらどうなるか確かめてみましょう）。ls -l の出力の各行の各項目は複数の空白によって区切られています。そのため連続する空白をまず$_ =~ s/ */ /g;によって１つの空白にします。これは「空白の次にさらに空白が続いたら、１つの空白にせよ」、という置換になります。そして split で空白をもとに行を分割し、ここから８番目の情報を取り出します。ls -l では多くの場合、７番目、もしくは８番目の情報がファイル名に対応します。

注意：ls –l の出力形式によっては上記のプログラムは正常に動かない場合があります。この場合は動くように自力で直してみましょう。

ここでは詳しくは触れませんが、逆にコマンドに対して文字列を入力するには、

```
open(FILE, "| UNIX コマンド");
```

とした後、

```
print FILE “コマンドに対する入力”;
```

とします。

課題 14-1：ls –l の出力を加工し、

```
The size of file ファイル名 1 is ファイル 1 の容量 bytes
The size of file ファイル名 2 is ファイル 2 の容量 bytes
The size of file ファイル名 3 is ファイル 3 の容量 bytes
                    :
                    :
```

と出力するようにしてみましょう。

15. リファレンス

「リファレンス」は Perl で扱えるデータ形式の一つで、日本語で言うところの「参照」にあたります。「リファレンス」は日本語で「参照」と言われる場合もあります。

通常、プログラム中では変数そのものの値を渡したり受け取ったりします。これは非常に簡単なのですが、実際には計算コスト(現実的にはプログラムの実行時間)がかかる場合が出てきます。

そのような場合に利用できるのが「リファレンス」です。リファレンスを利用することによって、大きなデータを繰り返し扱うような場合に計算コストを圧縮することができるようになります。

ここではリファレンスの概念から、その簡単な使い方までを解説します。

15.1 値渡しとの比較

通常、Perl で関数に値を渡す場合には「値渡し」という方法が使われます。

```
$data = "Welcome to the Perl world.";
&func($data);
exit;

sub func()
{
  $d = $_[0];

  print substr($d, 10, 10);
}
```

値渡しを利用すると、プログラム作成者の意図とは関係なく、暗黙のうちに渡したい値をプログラム内部で「コピー」してオリジナルと同じものを作成して関数に渡します。

それに対して「リファレンス」を利用すると、関数に渡す時に内部コピー作成してそのコピーを渡すのではなく、その値を格納してある「場所」を渡すことになります。C 言語を知っている人には「ポインタと同じ」と言えば分かりやすいでしょうか。

イメージとしては「太郎さんの家ってどんなん?」と聞かれた場合に、

・もう一つ太郎さんの家と同じものを作って「これ」と教える
・実際の太郎さんの家を「あれ」と指差す

といったところでしょうか。明らかに後者の方が時間的に楽なのが分かると思います。前者が値渡し、後者がリファレンス渡しです。

このことはどのような意味を持つのでしょうか。具体的に以下のような場合を想定してみましょう。

例として、「$seq」という変数に 100,000 文字分の a,t,g,c の並んだ文字列が格納されてい

るとします。これを count_gc() という関数に渡すことを考えてみましょう。

プログラムとしては、

```
#!/usr/bin/perl

sub count_gc(){
  my $data = $_[0];
  my $num = 0;
  my $i;
  for($i = 0;$i < length($data); $i ++){
     if(substr($data, $i, 1) eq "c" ||
        substr($data, $i, 1) eq "g"){ $num ++; }
  }  # ここは、$num = $data =~ tr/cg/cg/; としてもよい。

  return $num;
}

my $seq = "atcg" x 25000; # “atcg”を 25,000 回つなげる。
my $n = &count_gc($seq);

print "Number of GC: $n¥n";
```

のようになります。この場合、値渡しを利用しているので、$seq というデータとまったく同じデータをプログラム内部で複製して count_gc() に渡していることになります。

それでは count_gc() に渡す配列データを少しずつ変更して何度も繰り返したい、という場合にはどうなるでしょうか。

例えば、もともとの文字列情報が 100,000,000 文字分あって、それを 1,000 個に分割して count_gc() でそれぞれを計算して結果を合計するとします。

そのような場合、count_gc() が 1,000 回呼ばれることになり、結果、値渡しの際の「内部コピー」が 1,000 回行われることになります。

この「内部コピー」にかかる時間は、短い配列情報だったり繰り返しの回数が少なかったりするとあまり目立ちませんが、扱う情報が大きくなればなるほどより多くの時間がかか

るようになります。

実際、「100,000 文字のデータを 1,000 回関数に渡す」だけで、試したところ 13 秒もかかってしまいました(Celeron 300MHz の Linux マシンを使用)。配列の解析などはしておらず、「ただ関数に渡すだけ」のプログラムなのに、1,000 回繰り返すとこれだけ時間がかかってしまうのです。

しかし、関数にデータを渡すだけのプログラムをリファンレンスを使って実行してみたところ、わずか 0.3 秒で終了したのです。

つまり、「100,000 文字の配列データを関数に 1,000 回渡す」というだけのことを達成するのに、値渡しの場合には 13 秒もかかり、リファレンス渡しの場合には 0.3 秒で済んだということになります。

膨大なデータを扱わなければならない場合、この「内部コピー」の時間が無視できないほど大きなものになることは容易に想像できるでしょう。リファレンスを利用することによって、無駄なデータの内部コピー時間を大幅に短縮することができるようになります。

15.2 リファレンスの実際

それでは実際にリファレンスを使ってみましょう。

データを関数にリファレンスで渡すには、通常は関数呼び出しを

```
&func($data);
```

としているところを

```
&func(¥$data);
```

とするだけでいいのです。

対する関数の中では、リファレンスで渡されたデータを利用するには

```
sub func()
{
  my $ref = $_[0];

  print $$ref;
}
```

とします。通常の場合と違い、「$ref」という変数の前にもうひとつ「$」が付いていることに注意してください。

基本的には、以下の 2 つをペアにして覚えればリファレンスをつかいこなすことができるようになります。

- リファレンスを利用するには変数名の前に「¥」を付ける
- リファレンスの中味(変数の値)を利用するには「$」を付ける。ただし、リファレンスの中味が配列の場合は「@」、ハッシュの場合は「%」を使う。

下記の例を見てみましょう。

```
#!/usr/bin/perl

sub use_ref {
	my $ref1 = $_[0];
	my $ref2 = $_[1];
	my $ref3 = $_[2];

	print "$$ref1¥n";
	print "$$ref2[2]¥n";
	print "$$ref3{ Eigo }¥n";
}

my $scalar = "Hello";
my @array = ("Sun", "Mon", "Tue", "Wed", "Thr", "Fri", "Sat");
my %hash = ("Eigo" => "English", "Sugaku" => "Mathematics");

use_ref(¥$scalar, ¥@array, ¥%hash);
```

課題 15-1 先ほど出てきた cg の数をカウントするプログラムをリファレンスを利用するように書き直しましょう。

16. 変数の宣言チェック

C 言語などとちがって Perl では変数の宣言をする必要がありません。しかし、これまでみてきたような my などを使って変数の宣言をした方がプログラムの再利用性と解読性が向上します。

プログラムの先頭を以下のようにすることによって、宣言されていない変数や、初期値が入っていない変数のチェックが行われるようになります。

```
#!/usr/bin/perl –w

use strict;
```

基本的には、my を使って全ての変数の宣言をすれば OK です。
ファイルハンドルは local *を使って宣言します。

```
#!/usr/bin/perl –w

use strict;

my($scalar, @array, %hash);
local(*FILE);
```