ver 20170927

石浦研究室「Perl プログラミング」(©石浦 菜岐佐 2005~ 2017, 福本 貴之 2013, 中村 遼 2013)

1 Perl の基礎

1.1 Perl の実行

Perl 版の “Hello world“ は次の通り.

```
print "Hello world\n";
```

これを hello.pl というファイルに格納したとすると,

```
perl hello.pl
```

で実行できる. Unix や Cygwin なら,

[List 1.1]

```
#!/usr/bin/env perl
print "Hello world\n";
```

のように hello.pl の 1 行目で env を使って Perl のパスを指定し, hello.pl をユーザ権限で実行可能にしておけば[1] ,

```
./hello.pl
```

だけで起動できる.

[List 1.1] を作成し, 実行してみよ.

1.2 Perl のスカラ変数

- Perl の変数 (スカラ変数) は先頭に $ をつける. 変数を使ったスクリプトの例を示す.

[List 1.2]

```
1:  #!/usr/bin/env perl
2:  $price = 198000;
3:  $count = 9;
4:  $item = "PC";
5:  print "$item 1 台 $price 円\n";
6:  print "$count 台買うと " . ($price*$count) . " 円\n";
7:  $price = "$count 台以上買ってくれたらタダ";
8:  print "でも $price\n";
```

を実行すると,

```
PC 1 台 198000 円
9 台買うと 1782000 円
でも 9 台以上買ってくれたらタダ
```

と表示される.

[1] コマンドは例えば chmod +x hello.pl

– 変数名は, 先頭に $ が付く以外は C 言語と同じ (変数名は, 英字かアンダースコア (_) で始まる英数字列. 大文字と小文字は区別する).
– 変数は**宣言不要**. (ただし, 後述のように宣言する方法がある.)
– 変数には**型がない**ので, 整数でも文字列でも浮動小数点数でも何でも代入できる. 7 行目にあるように, 一旦整数を代入した変数に後から文字列を代入することも可能.
– 出力は, `print` の後ろに出力したい項目を . (文字列連結演算子) で切って並べればよい. また, ダブルクォート (") で囲んだ文字列の中には変数を直接書くことができる.

1.3 値

1.3.1 ”abc” と ’abc’

Perl の文字列は, ダブルクォートだけでなくシングルクォート (’) も用いることができる.

– 次では `$foo` と `$bar` には同じ文字列が代入される.

```
$foo = "abc";
$bar = 'abc';
```

– ダブルクォートとシングルクォートには大きな違いがある.

```
$price = 19800;
print "1 個 $price 円 \n";
print '1 個 $price 円 \n';
```

を実行すると

```
1 個 19800 円
1 個 $price 円 \n
```

と表示される. ダブルクォートで囲うと変数などの評価が行われるのに対し, シングルクォートで囲うと文字列が本当にそのまま表示される. `\n` さえそのまま表示されて改行記号にならない.

1.3.2 値の変換

Perl は型に関しては結構いい加減 (に感じられる).

– 文字列から数値への変換は自動で行われるので, 特に意識しなくてもよい.

```
$foo = "123";
$bar = "111.111";
$baz = $foo + $bar;
```

と C では許されないような文でも, `$baz` にはちゃんと `234.111` が代入される. (ちなみに, 数値に変換できないような文字列は `0` に変換される.)

– 数値についても int/double のような区別はない.

```
$foo = 1/3;
```

は自動的に小数点数に変換される. 強制的に整数化したければ, 整数化関数 `int` を用いて

```
$foo = int(1/3);
```

とする.

– 変数の初期値には「undef(未定義値)」が入っていて, defined 関数で変数が定義されているかどうか調べることができる. 初期化忘れの変数 $foo を -1 で初期化したい場合は, 次のように書けばよい.

```
if (!defined $foo) {$foo = -1;}
```

あるいは, if の逆の評価[2] を行う unless を使って, このように書くこともできる.

```
unless (defined $foo) {$foo = -1;}
```

なお, 定義されていない変数は, 数値の文脈では 0, 文字列の文脈では '' (空の文字列) と解釈される.

1.3.3 use strict と my による変数宣言

変数宣言しなくていいのは便利だが, 思わぬミスが生じる. 例えば, 変数名をタイプミスしても何のエラーも出ずに実行されてしまい, デバッグが困難になることがある. そこで, 次の例のように, **変数を必ず宣言し, 宣言していない変数をチェックする設定にする**ことを推奨する.

[List 1.3]

```
1:  #!/usr/bin/env perl
2:  use strict;    # 未宣言変数を許さない等, 厳密な書式を強制する
3:  use warnings;  # 未定義変数の使用等, 動的チェックを行い警告する
4:
5:  my $foo = 10;  # my はローカル変数の宣言
6:  my $bar = 20;
7:  $foo *= $bar;
8:  my $baz;
9:  $baz = $foo + $bar;
```

– 最初の 3 行は決まり文句としてファイルに持っておこう.
– my がローカル変数の宣言[3]. 型はないので, ローカル変数であることのみを宣言する. C++ と同じように, 使用前ならどこに書いてもよい. 基本的には, 5, 6 行目のように宣言と同時に初期化する書き方を推奨する.
– 複数のローカル変数を一行で宣言&初期化したいときは, 次のように書くこともできる.

```
my ($foo, $bar) = (10, 20);
```

【EX-1.1】 [List 1.2] に use strict; と use warnings; を設定し, 必要な変数宣言を追加して実行できるようにしたスクリプト vardef.pl を作成せよ.

1.4 制御構造

基本的に C と同じだが, 少しづつ違うところがあるので, それを簡単にまとめる.

1.4.1 条件分岐とループ

if 文の違いは次の 2 点

– 条件成立時に実行する文や else に続く文は, 1 文であっても**必ず { } で囲う**ことが必要.

[2]条件式が偽なら処理を行う.

[3]「変なキーワード」と感じる人は多いはず. 実は Perl 4 でローカル変数を作るための仕組みがあってそのキーワードが local だったが, 効率の良い仕組みではないので, Perl 5 で新たな仕組みができた. 同じキーワードは使えないのでこのようなキーワードになったのだろう. ちなみに, グローバル変数を宣言するキーワードは our だったりする.

– **else if の代わりに elsif を用いる.**

C と Perl を比較すると, 次のとおり.

```
if (foo > 0) sign = '+';
else if (foo < 0) sign = '-';
else sign = ' ';
```

```
if ($foo > 0) {$sign = '+';}
elsif ($foo < 0) {$sign = '-';}
else {$sign = ' ';}
```

Perl には **switch-case はない**[4] ので, `if` ～ `elsif` ～ ··· ～ `else` で代用する.

while 文と do–while 文は, ループ本体が 1 文でも必ず { } で囲まなければならない以外は C 言語と同じ.

`for` も, ループ本体を必ず { } で囲まなければならない

– ループ制御変数は, 下の例のように for 文の ( ) 内で `my` 宣言する.
– C の `break` は Perl では `last` になる.

C と Perl を比較すると, 次の通り. ただし, Perl ではこのような for 文を書く機会はほとんどない[5].

```
for (i = 2; i <= 100; i++) {
  if (i % 8 == 0 && i % 17 == 0) break;
}
```

```
for (my $i = 2; $i <= 100; $i++) {
  if ($i % 8 == 0 && $i %17 == 0) {last;}
}
```

文字列の比較には辞書順で文字列を比較する演算が用意されている.

–

文字列比較演算	数値比較演算
eq	==
ne	!=
lt	<
le	<=
gt	>
ge	>=
cmp	<=>

`cmp` や `<=>` は次のように `-1`, `0`, `1` を返す 3-way の比較演算である (ソーティング演算等で用いる).

$$\texttt{\$a<=>\$b} \begin{cases} \texttt{\$a<\$b} & \text{なら } \texttt{-1} \\ \texttt{\$a==\$b} & \text{なら } \texttt{0} \\ \texttt{\$a>\$b} & \text{なら } \texttt{1} \end{cases}$$

条件の式に関する注意

– C 言語と同様

```
if ($foo != 0) {print "non-zero\n";}
```

は次のようにも書ける.

```
if ($foo) {print "non-zero\n";}
```

[4] Perl 5.10 以降なら, `switch-case` と同じ働きをする `given-when` が用意されている.
[5] 後述の「レンジ」を利用することで, 簡単に書くことができる.

– 長さ 0 の文字列 '' は, if 文等の条件式の文脈では 0 と評価されるので

```
if ($foo ne '') {print "non-null string\n";}
```

は

```
if ($foo) {print "non-null string\n";}
```

とほぼ同じ意味になる. ほぼというのは, $foo が '0' の場合には, 文字列が空ではないのに 0 と評価されてしまうからである. 「文字列が空でなければ」を表現したいときには $foo ne '' と明示的に書く方が安全.

【EX-1.2】 入力した整数を素因数分解して (素因数をスペースで区切って) 出力するスクリプト factorize1.pl を作成せよ. 変数 $n への整数値 1 個の入力は chomp(my $n = <STDIN>); により行える (下記のスクリプトを参考にせよ). もし平方根が必要であれば, 組込み関数 sqrt(値) を用いよ. 次の入出力が得られることを確認せよ.

入力	期待値
10	2 5
3	3
8	2 2 2
17640	2 2 2 3 3 5 7 7
9973	9973

```
#!/usr/bin/env perl
use strict;
use warnings;

my $n;
warn "$n = ";  # 標準エラー出力にメッセージ出力
# <STDIN> は標準入力からの 1 行の入力で, chomp は行末の改行記号の除去
chomp($n = <STDIN>);
print "n = '$n'\n";    # 読み込んだデータを出力
```

【EX-1.3】 文字列 $str に適当な英単語を初期設定し, これを当てさせるゲームを行うスクリプト wguess.pl を作成せよ. 推定単語を入力する度にその単語が $str よりも辞書順で前ならば 'before', 後ならば'after' と表示し, 一致すれば 'congratulations!' と表示して終了するようにせよ. 推定単語 $w の入力方法は前の課題と同じである. なお, 動作確認の都合上, $str の初期値は 'perl' として提出せよ.

1.4.2 関数

- Hello とプリントする関数 hello とその呼出しは, 次のように書ける.

```
 1:  #!/usr/bin/env perl
 2:  use strict; use warnings;
 3:
 4:  hello();
 5:  &hello(); # 先頭に & をつけてもよい
 6:
 7:  sub hello # 関数の定義は sub で始まる
 8:  {
 9:    print "Hello\n";
10:  }
```

- 関数の定義 (本体) より前に呼出しがあってもかまわない.
- 関数呼出しの & は文脈から関数と分かる場合は省略できる. また, 引数が無い場合は () は省略できる. しかし, 両方を省略することはできない.
- モダンな Perl では, &を省略し, () を省略しない形で関数を呼び出すことが多い.

- 引数や返り値がある場合は次の通り. 2 数の平均を返す関数 ave とその呼出しの例を示す.

[List 1.4]

```
 1:  #!/usr/bin/env perl
 2:  use strict; use warnings;
 3:
 4:  my $foo = 10;
 5:  my $bar = 15;
 6:  my $baz = ave($foo, $bar+5);   # 関数の呼出し
 7:  print "baz = $baz\n";
 8:
 9:  sub ave                    # 関数の定義
10:  {
11:    my ($x, $y) = @_;        # 引数の受け取りはこのように書く
12:    my $foo = ($x + $y) / 2;
13:    return $foo;
14:  }
```

- 11 行目の, 関数に与えられた引数の受け取り方は, この時点では「決まった書き方」として覚えておこう.

ファイルは複数に分割することもできる. 関数群だけが入ったファイルを準備し, メインのスクリプトからその使用を require 文で宣言する, という方法が一般的である. [List 1.4] の ave 関数を ave.pl というファイルに格納し, これをメインのスクリプトから使用すると, 次のようになる.

[List 1.5]

```
1:  #!/usr/bin/env perl
2:  use strict; use warnings;
3:
4:  require "ave.pl";    # ave.pl 内の関数を使用することを宣言
5:
6:  my $foo = 10;
7:  my $bar = 15;
8:  my $baz = ave($foo,$bar+5);
9:  print "baz = $baz\n";
```

ave.pl (関数だけを格納したファイル)

```
 1:  use strict; use warnings;
 2:
 3:  sub ave
 4:  {
 5:    my ($x,$y) = @_;
 6:    my $foo = ($x+$y)/2;
 7:    return $foo;
 8:  }
 9:
10:  1;  # require されるスクリプトは，最後に真値を返さなければならない
```

【EX-1.4】 【EX-1.2】 の素因数分解を行う部分を関数として独立させよ. 即ち, 整数を 1 つ受け取って素因数分解して表示する部分を関数 factorize を作成し, これを factorize2.pl に格納せよ. 次の factorize2main.pl を実行して, 【EX-1.2】 と同じ結果が得られることを確認せよ.

```
#!/usr/bin/env perl
use strict;
use warnings;

require "factorize2.pl";

warn "n = ";
chomp(my $n = <STDIN>);
factorize($n);
```

2 配列と連想配列

2.1 配列

2.1.1 配列の記法

- 変数名の前に, $ ではなく @ を付けると配列になる[6]. 配列の各要素は $foo[式] のように参照する.

  [List 2.1]

```
 1:  #!/usr/bin/env perl
 2:  use strict;
 3:  use warnings;
 4:
 5:  my @array;          # 配列の宣言（サイズは宣言しなくてよい）
 6:
 7:  $array[0] = 1;      # 各要素への代入
 8:  $array[1] = 3;
 9:  $array[2] = 'abc';  # 型がないので，1 つの配列内に数と文字列が混在してもよい
10:  $array[3] = 6;
11:
12:  for (my $i = 0; $i < 4; $i++) {   # 各要素の表示
13:    print "\$array[$i] = $array[$i]\n";
14:  }
```

  – Perl の配列は必要に応じて**動的にサイズが代わる**ので, 宣言時 (配列を作るとき) にサイズを定義する必要はない.

  – 配列 @array のサイズ (要素数) は scalar @array で求められる. 従って, 12˜ 14 行目は次のように書くこともできる.

```
12:  for (my $i = 0; $i < scalar @array; $i++) {
13:    print "\$array[$i] = $array[$i]\n";
14:  }
```

  – 配列 @array の最後の要素の番号 (添字) は $#array で求められるので, 次のように書くのがベター.

```
12:  for (my $i = 0; $i <= $#array; $i++) {
13:    print "\$array[$i] = $array[$i]\n";
14:  }
```

- 配列のデータは ( ) を使ったリスト (スカラの集合) で表現することができ, 配列の初期化や代入で使える (非常に便利).

  [List 2.1] は次のように書ける.

[6] スカラ変数, 配列, 後述する連想配列で同じ名前が使われた場合, これらは区別される.

```
1:  #!/usr/bin/env perl
2:  use strict;
3:  use warnings;
4:
5:  my @array = (1, 3, 'abc', 6);   # 配列の宣言と初期化
6:
7:  for (my $i = 0; $i <= $#array; $i++) {
8:    print "\$array[$i] = $array[$i]\n";
9:  }
```

– また, ( ) の中には配列を書くこともでき, 配列への要素の追加や配列の連結も容易に書ける.

[List 2.2]

```
 1:  #!/usr/bin/env perl
 2:  use strict; use warnings;
 3:
 4:  my @array1 = (1, 2, 3, 4);
 5:  my @array2 = ('a', 'b', 'c');
 6:
 7:  @array1 = (@array1, 5, 6);   # @array1 = (1,2,3,4,5,6) となる
 8:  @array1 = (0, @array1);     # @array1 = (0,1,2,3,4,5,6) となる
 9:  @array2 = (@array2, @array2);    # @array2 = ('a','b','c','a','b','c') となる
10:
11:  # 配列はそのままプリントすることも可能（便利）
12:  print "@array1\n";   # 0 1 2 3 4 5 6 が出力される
```

【EX-2.1】 下記の空欄を埋め, 配列のデータを数値の昇順にソートする sort.pl を完成させよ (後に述べる `sort` 関数を用いないこと).

```
#!/usr/bin/env perl
use strict; use warnings;

my @array = (9, 5, 7, 1, 3, 4, 6);

# ここを埋めて, @array のデータがソートされるようにせよ

print "\@array = (@array)\n";   # @array = (1 3 4 5 6 7 9) が出力される
```

2.1.2 配列の操作

配列の操作に便利な記法や組込み関数がある.

1. `push`, `pop`, `shift`, `unshift`

`push(@array, $data)`	配列 `@array` の末尾に $data を追加する
`unshift(@array, $data)`	配列 `@array` の先頭に $data を追加する
`pop(@array)`	配列 `@array` の末尾の要素を取り出す
`shift(@array)`	配列 `@array` の先頭の要素を取り出す

– 例えば, 配列に次々にデータを投入して行く際には push を用いる.

```
my @array;
push(@array, 1);
push(@array, 3);
push(@array, 'abc');
push(@array, 6);
# 結果 @array = (1, 3, 'abc', 6) となる
```

– 配列データを先頭から次々に読み出して処理する際には shift を用いる.

```
while(@array) {   # @array が空になると 0 と評価され, 終了する
  my $next = shift(@array);
  print "次の要素は $next \n";
}
```

– push と pop の組合せでスタックが実現できる.

– push と shift の組合せでキューが実現できる.

2. join, split

join('区切り記号', @array)	@array を区切り記号で連結して文字列にする
split(/区切りパターン/, $string)	$string を区切りパターン (正規表現) で分割し, 配列にする

– join を用いると, 配列要素の表示が容易にできる.

```
my @array = (1, 2, 3, 4, 5);
my $str = join(',', @array);
print "\@array = ($str)\n";
# @array = (1,2,3,4,5) と表示される
```

– split を用いると, 文字列の分断が容易にできる. /区切りパターン/ には単純な文字だけでなく「正規表現」が書ける (詳細は後述).

```
my $str = "00:09:23:A3:B4:FF";
my @array = split(/:/, $str);
# @array = ('00', '09', '23', 'A3', 'B4', 'FF') となる.
```

– また, 次のような使い方で文字列からデータの抽出が容易にできる.

```
my $line = "論理回路:1 年生:春学期:2:石浦";
my ($subject, $grade, $semester, $credit, $prof) = split(/:/, $line);
```

【EX-2.2】 前の演習で作成した素因数分解のスクリプト factorize2.pl を次のように書き換えよ (factorize3.pl とせよ).

– 関数 factorize 内で, @factor という配列を宣言し, 新しい素因数が見つかる度に push を使って @factor に追加する.

– 結果の表示を join を用いて行え (@factor = (2, 2, 3) なら 2*2*3 と表示せよ).

factorize2main.pl の require "factorize2.pl"; を require "factorize3.pl"; に書き換えたスクリプトを実行し, 結果を確認せよ.

3. for

– 配列要素に対する繰り返しには, for 文の次のような記法が使える.

```
my @professors = (' 浅野', ' 井坂', ' 石浦', ' 岡田');
for my $professor (@professors) {
  print "$professor 先生 \n";
}
```

- なお, for を foreach と書くこともできる (for と foreach は等価).
- 基本的に, for と書いておけば OK.

4. sort
   - 一文で配列要素のソーティングを行うことができる.

```
my @array = ('America', 'Japan', 'Korea', 'France', 'Germany');
@array = sort @array;
```

   - デフォルトの比較順序は「文字列として辞書順」である. 整数の比較の場合には比較関数を陽に指定する必要がある. 比較関数は, $a, $b という変数に対して指定する. (このため, この手のソートを行う場合には, 同じスコープで変数 $a, $b を別目的で使用しないこと.)

```
my @array = (1, 23, 15, 7, 11, 3);

@array = sort @array;
# 文字列としてのソート． @array = (1, 11, 15, 23, 3, 7) となる

@array = sort { $a <=> $b } @array;
# 数字としてのソート． @array = (1, 3, 7, 11, 15, 23) となる

@array = sort { $b <=> $a } @array;
# 数字として逆順のソート． @array = (23, 15, 11, 7, 3, 1) となる

@array = sort { $b cmp $a } @array;
# 文字列として逆順のソート． @array = (7, 3, 23, 15, 11, 1) となる
```

   - 複雑な比較も行える.

```
my @array = (1, 23, 15, 7, 11, 3);
@array = sort { $a % 10 <=> $b % 10 } @array;
# 10 で割った余り (つまり 10 進数の 1 桁目) でソート.
# @array = (1 11 23 3 15 7) 等となる (1 桁目が等しい場合の順序は処理系依存).

@array = sort { $a % 10 <=> $b % 10 || $a / 10 <=> $b / 10 } @array;
# 1 桁目でソートし，等しい場合は上位桁でソート.
# 比較の結果が等しいときには || の左辺が 0 になるので，右辺が評価される
# @array = (1 11 3 23 15 7) となる.
```

【EX-2.3】 下記の空欄を埋め, 配列内のデータ (文字列) を, 長さの昇順にソートし, 長さが等しければ辞書順にソートするスクリプト strlensort.pl を作成せよ. 文字列 $s の長さは length($s) により求められる. 表示の際には for を用いて, 1 行に 1 文字列づつ表示するようにせよ.

```
#!/usr/bin/env perl
use strict; use warnings;

my @array = qw/this that an it are apple/;
# my @array = ('this', 'that', 'an', 'is', 'are', 'apple'); の別記法
# 半角スペースが配列の区切りとなる.

# @array のデータがソートされるようにせよ
# ソートされたデータを, 1 行に 1 文字列づつ出力せよ
```

下記が出力されることを確認せよ.

```
an
is
are
that
this
apple
```

5. レンジと繰り返し
   - 配列の初期化などに便利な記法がある.

```
my @array1 = (1..10);
# レンジ
# @array1 = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) となる.

my @array2 = reverse (1..10);
# レンジ + 逆順
# @array2 = (10, 9, 8, 7, 6, 5, 4, 3, 2, 1) となる.

my @array3 = 3 x 5;
# 繰り返し
# @array3 = (3, 3, 3, 3, 3) となる.
```

   - レンジを使うことで, for 文を簡潔に書くことができる.
     例えば, [List 2.1] は次のように書ける.

```
1:  #!/usr/bin/env perl
2:  use strict;
3:  use warnings;
4:
5:  my @array = (1, 3, 'abc', 6);
6:
7:  for my $i (0..$#array) {
8:    print "\$array[$i] = $array[$i]\n";
9:  }
```

6. grep, map

grep { 式 } @array	@array の中から「式」を満たすものだけ取り出した配列を作る
map { 式 } @array	@array の各要素を「式」で操作する

「式」中では, $_ という変数が各要素を表す.

– 配列の中から正の数だけ取り出す.

```
my @array = (1, -3, 4, -9, -6, 2, 8);
@array = grep { 0<$_ } @array;
# @array = (1, 4, 2, 8) となる
```

– 配列の要素をすべて 2 乗する.

```
my @array = (1, 2, 3, 4, 5);
@array = map { $_*$_ } @array;
# @array = (1, 4, 9, 16, 25) となる
```

【EX-2.4】 レンジと grep と map を用いて, 1 から 19 までの奇数の 2 乗からなる配列を作成し, それを (1,9,25,49,81,...) のような形式で出力するスクリプト oddsq.pl を作成せよ.

2.1.3 配列と関数

- 関数の引数の受け取りに出て来た @_ は, 引数を格納する配列である.
  – 2 引数の平均値を計算する関数 ave

```
sub ave
{
  my ($x, $y) = @_;
  my $foo = ($x + $y) / 2;
  return $foo;
}
```

は次と等価である.

```
sub ave
{
  return ($_[0] + $_[1]) / 2;
}
```

関数呼出しの引数に配列を書けば, その配列の要素が順に @_ に代入されて関数に渡される.

–

```
my $foo = 2003;
my @array1 = (1, 2, 3, 4);
my @array2 = ('a', 'b', 'c');
func($foo, @array1, @array2);

sub func
{
  print "arg = (" .  join(',', @_) .  ")\n";
}
```

を実行すると,

```
2003,1,2,3,4,a,b,c
```

が表示される. このように, 配列の引き渡し方は C 言語とは異なる.

要素が値渡しで渡されるため, 次のような問題が生じる.

* 巨大な配列を渡すとき効率が悪い.

* 2 つ以上の配列を渡すとき, 受け取る側ではその切れ目が分からない (上の例で, どこまでが @array1 でどこまでが @array2 か分からない).

これは, C 言語のポインタに相当するリファレンスを使った引数の受渡し (後述) で解決できる.

- return 文に配列 (あるいはスカラ値のリスト) を書けば, 関数から呼出し元に配列を返すことができる.
  - 例えば,

```
sub map_square
{
  my @ret;
  for my $i (@_) {
    push(@ret, $i ** 2);
  }
  return @ret;
}

my @array = map_square(1, 2, 3, 4, 5);
```

を実行すると, @array = (1, 4, 9, 16, 25) となる.

  - 関数が複数の値を返すことができるので便利である. 次は, フルネームとパスワードを含む 'Nagisa Ishiura:Aikotoba' のような文字列から, ファーストネーム, ラストネーム, パスワードを抽出する関数である.

```
my ($first_name, $last_name, $passwd) = user_info('Nagisa Ishiura:Aikotoba');
# $first_name = 'Nagisa'
# $last_name = 'Ishiura'
# $passwd = 'Aikotoba'

sub user_info
{
  my ($line) = @_;
  my ($full_name, $passwd) = split(/:/, $line);
  my ($first_name, $last_name) = split(/ /, $full_name);
  return ($first_name, $last_name, $passwd);
}
```

【EX-2.5】 【EX-2.2】の factorize3.pl を次のように書き換えよ.

- 関数 factorize は, 関数内で素因数分解の結果を表示するのではなく, 得られた素因数の配列を返すようにせよ.
- 素因数の配列を受け取って表示を行う関数 print_factors を作成せよ.

これら 2 つの関数を factorize4.pl に格納し, 次の factorize4main.pl を実行して, 動作を確認せよ.

```
#!/usr/bin/env perl
use strict;
use warnings;

require "factorize4.pl";

# print_factors のテスト
print_factors(1..5);

warn "n = ";
chomp(my $n = <STDIN>) ;
my @factors = factorize($n);
print_factors(@factors);
```

2.2 連想配列

2.2.1 連想配列とは

- 連想配列とは, いわば文字列をキーとする配列であり, Perl を非常に強力なものにしている.
- 変数名の前に % を付けると連想配列になる. 連想配列の各要素は $a{ 式 } のように参照する.

[List 2.3]

```
1:  my %city;                          # 連想配列の宣言
2:
3:  $city{'兵庫県'} = '神戸市';        # 各要素の代入
4:  $city{'大阪府'} = '大阪市';
5:  $city{'滋賀県'} = '大津市';
6:
7:  for my $pref ('兵庫県','大阪府','滋賀県') {
8:    print "$pref の府県庁所在地は$city{$pref} です \n";
9:  }
```

を実行すると

```
兵庫県の府県庁所在地は神戸市です
大阪府の府県庁所在地は大阪市です
滋賀県の府県庁所在地は大津市です
```

と表示される.

– 上の例では, キーにクォーテーションを使っているが,
  * 整数
  * 固定小数点数
  * 変数名として許される英数字列

  の場合はクォーテーションは省略できる.

```
my %foo;
$foo{Japan} = 'Tokyo';
$foo{123} = '1';
```

– かつて連想配列は「遅い」と言われたが, ハッシュを用いて実装されているため, 比較的高速である.

連想配列は ( ) を使って初期化や代入ができる.

– [List 2.3] の%foo への代入は, 次のようにキーと値のペアを列挙することにより行える.

```
my %city = (’兵庫県’,’神戸市’, ’大阪府’, ’大阪市’, ’滋賀県’,’大津市’);
```

しかし, これだと切れ目が分かりにくいので, ファットカンマ (=>) を用いた次の記法も準備されている (この書き方を推奨する).

```
my %city = (’兵庫県’ => ’神戸市’, ’大阪府’ => ’大阪市’, ’滋賀県’ => ’大津市’);
```

– キーが変数名として許される英数字列の場合は, ファットカンマを使うことで, 次のようにキーのシングルクオートを省略することができる.

```
my %city = (hyogo => ’神戸市’, osaka => ’大阪市’, shiga => ’大津市’);
```

2.2.2 連想配列の操作

連想配列そのものが強力なので操作は多くない.

keys %hash	%hash のすべてのキーの配列を返す
values %hash	%hash のすべての値の配列をする
each %hash	キー値、値の一組を返す
exists $hash$key	%hash 中にキーが $key であるデータが登録されているかどうかを返す
delete $hash$key	%hash からキーが $key であるデータを削除する.

重要なもののみ補足する

1. keys

   連想配列の全データを取り出す際に用いる. [List 2.3] の例は, 次のように書くのがベター.

   [List 2.4]

```
1:  my %city = (
2:    ’兵庫県’ => ’神戸市’,
3:    ’大阪府’ => ’大阪市’,
4:    ’滋賀県’ => ’大津市’, # 最後のコンマはエラーではない
5:  );
6:
7:  for my $pref (keys %city) {
8:    print "$pref の府県庁所在地は$city{$pref} です \n";
9:  }
```

   4 行目の最後コンマはエラーではない (エラーにならない). データの追加削除が編集しやすいよう, 最後の要素の後ろにコンマを書くことが許されている.

   keys が返す配列は**順序がランダム**で, データの登録順にもキーの昇順にもなっていない. これは, 連想配列がハッシュで実装されているためである. キーの昇順にしたければ, 9 行目を

```
7:  for my $pref (sort keys %city) {
```

   と書けばよい.

   keys は実際にキーの配列を作るので, 超巨大な連想配列を操作したいときには記憶効率が悪い. このような場合には, 1 つづキーと値の組を取り出す each を用いる.

```
 7:  while (my ($pref,$city) = each %city) {
 8:    print "$pref の府県庁所在地は$city です \n";
 9:  }
```

【EX-2.6】 文字列にスペースで区切られた英単語列を受け取り, 各単語の出現回数を数えて辞書順に結果を表示する関数 wdist を作成せよ. 下記リストを完成させて動作確認し, `wdist.pl` として提出せよ.

```
#!/usr/bin/env perl
use strict;
use warnings;

my $s = 'this is a pen that is a pencil that he used';
wdist($s);

sub wdist {
  # ここを埋めて完成させよ
}
```

これを実行して, 次の出力が得られるようにすること.

```
a: 2
he: 1
is: 2
pen: 1
pencil: 1
that: 2
this: 1
used: 1
```

2.2.3 連想配列と関数

- 連想配列を関数の引数として渡す方法は, 次の通り.

```
my %city = (
  '兵庫県' => '神戸市',
  '大阪府' => '大阪市',
  '滋賀県' => '大津市', # 最後のコンマはエラーではない
);
assoc_print(%city);

sub assoc_print
{
  my (%city) = @_;
  for my $pref (keys %x) {
    print "$pref の府県庁所在地は$city{$pref} です \n";
  }
}
```

  - ただし, この記法では連想配列が**値渡し**されるので, **要素数が多い場合には非常に効率が悪い**. このような場合には, 後述のリファレンスを用いて参照渡しにするべきである.

- 連想配列の値渡しの効果的な使用法は, キー (キーワード) を用いたパラメータの受渡しである.
  - 関数の引数が多かったり, デフォルトの引数が多い場合は, 次のような方法を用いるとよい.

[List 2.5]

```
 1:  print_date( year => 2003, month => 8, day => 31 );
 2:  print_date( month => 8, day => 31, year => 2003 ); # 順不同
 3:  print_date( year => 2003 ); # デフォルト
 4:
 5:  sub print_date
 6:  {
 7:    my %param = @_;
 8:    my $year  = defined $param{year}  ?  $param{year}  :  2002;
 9:    my $month = defined $param{month} ?  $param{month} :  1;
10:    my $date  = defined $param{date}   ?  $param{date}   :  1;
11:    print "$year-$month-$date\n";
12:  }
```

2.2.4 連想配列による構造体のエミュレーション

- Perl には構造体はない. 連想配列でまかなうのが標準的なやり方.
  - 例えば C 言語で

```
struct person {
  char name[32];
  char phone[64];
  int age;
} professor;

strcpy(professor.name, "石浦 菜岐佐")
strcpy(professor.phone, "079-565-7742")
p.age = 43;
```

と書くところを Perl で書けば,

```
my %professor = ();
$professor->{name} = "Nagisa Ishiura";
$professor->{phone} = "079-565-7742";
$professor->{age} = 43;
```

や

```
my %professor = (
  name => "Nagisa Ishiura",
  phone => "079-565-7742",
  age => 43,
);
```

等となる.

【EX-2.7】

人間の身長 (cm) と体重 (kg) をそれぞれ height, weight というキーを用いて連想配列で記憶するとする．このような連想配列を受け取って, BMI の値を返す関数 bmi を作成せよ．下記のリストの必要部分を埋めて, bmi.pl として提出せよ．

```
#!/usr/bin/env perl
use strict;
use warnings;

my %person1 = ( height => 170, weight => 60 );
printf "%6.2f\n", bmi(%person1);  # C と同じように printf が使える

my %person2 = ( height => 155, weight => 43 );
printf "%6.2f\n", bmi(%person2);

sub bmi
{
  # ここを埋めて完成させよ
}
```

下記の出力が期待される．

```
 20.76
 17.90
```

3　マッチングと文字列処理

マッチング演算は, 与えられた文字列中にある文字列パターン含まれるかどうかを判定するものである. これと同時に, 置換やマッチしたパターンを取り出すことができ, 連想配列と並ぶ Perl の強力な機能である.

3.1　マッチング

- 基本的なマッチングの記法は次の通り. 3 行目の =~ がマッチング演算子で, 右辺のスラッシュに囲まれた文字列がパターンである.

```
1:  my @prefs = qw/ Osaka Hyogo Kyoto Nara Shiga Wakayama /;
2:  for my $pref (@prefs) {
3:    if ($pref =~ /ka|yo/) {print "$pref\n";}
4:  }
```

このスクリプトでは, 配列の各要素が「ka または yo 」にマッチするかどうかを調べ, もしマッチすればそれを表示する. 実行すると,

```
Osaka
Hyogo
Kyoto
Wakayama
```

が表示される.

- マッチング演算の否定は !~ である. 上記リストの 3 行目を

```
3:    if ($pref !~ /ka|yo/) {print "$pref\n";}
```

にすれば, ka にも yo にもマッチングしないものだけが表示される.

```
Nara
Shiga
```

【EX-3.1】 入力した整数 $n に対し, 1 から $n までの数を文字列と見たときに'12' または '3' を含むものの個数を出力する, という動作を 8 回繰り返すスクリプト icount.pl を作成せよ. 下記の動作を確認せよ.

n	答え
12	2
30	5
40	14
100	20
120	24
123	27
1000	289

3.2　正規表現

- マッチング演算のパターンには「正規表現」が指定でき, 非常に多彩なものが表現できる.
- 正規表現には次の演算がある.

α | β	「α または β 」を表す. s(|ci)ence は, 「sence または science」を表す.
α β	「α の次に β が来る」を表す. (s|m)(it|an) は, {sit, san, mit, man} を表す.
α *	「α の 0 回以上の繰り返し」を表す. (an)* は, { 空列, an, anan, ananan, ···} を表す.
α +	「α の 1 回以上の繰り返し」を表す.
α ?	「α が 0 回か 1 回」を表す.
[$c_1c_2\cdots$]	「文字 c_1 または c_2 ···」を表す. [abcd] は {a, b, c, d} を表す. - を使って範囲を指定することができ, [a-z] は {a, b, c, ···, z}, [2-8] は {2, 3, 4, ···, 8} を表す. さらに, ^ で補集合をとることができ, [^a-z] は「アルファベットの小文字以外」を表す.

- この演算に次の特殊文字を組み合わせて使用できる.

\d	数字一文字. [0-9] と同じ.
\D	数字以外の文字.
\w	英数字.
\W	英数字以外の文字.
\s	空白文字 (スペースとタブ).
\S	空白文字以外の文字.
.	任意の一文字
^	文字列の先頭にマッチする.
$	文字列の末尾にマッチする.

- 【注意】正規表現の中でピリオドを表したい場合には \. と書かなければならない (さもなければ, 任意の一文字にマッチしてしまう). このように, 特別な意味を持つ次のような文字は, \ を前につけてエスケープしなければならない.
  - 正規表現の中で特別な意味をもつ文字 (. [ ] ( ) ^ $ 等)
  - 正規表現の始まりと終りを表すスラッシュ(/)
  - 変数に用いられる記号 (@ $ 等)
  - バックスラッシュ自身 (\ )
- いくつか例を示す.

^[a-zA-Z_]\w*	先頭が英字でその後に英数字が 0 個以上ある
\.pl$	.pl で終る
\d+(:\d+)+	コロンで区切られた数字列

- 正規表現はマッチングだけでなく, split 演算等でも用いる. split 演算の第一引数は正規表現である. 例えば, 文字列 $a から 1 つ以上の空白記号 (スペースまたはタブ) で区切られた単語を切り出すには, 次のように書けばよい.

```
my $str = 'This is  a pen  [タブ] That is an erasor ';
my @words = split(/\s+/, $str);
# @words = ('This','is','a','pen','That','is','an','erasor') となる
```

さらに, カンマやピリオドも含まれた文章から単語だけ抜き出すのは, 次で行うことができる.

[List 3.1]

```
my $str = 'This is a pen.  That, however, is an erasor!';
my @words = split(/\W+/, $str);
```

【EX-3.2】 文字列を入力し, それが正しい電子メールアドレスかどうかを判定して出力するという動作を 5 回繰り返すスクリプト mcheck.pl を作成せよ. 実は完全な処理系を作るのは難しいので, 次のテストケースで動作するものを作成せよ.

文字列	判定
abc12345@kwansei.ac.jp	○
ishiura@kwansei@ac.jp	×
Nagisa.Ishiura@ml.kwansei.ac.jp	○
nagisa_ishiura@asip-solutions.com	○
abc12345@ist..kwansei.ac.jp	×

3.3 マッチした文字列の切り出し

- マッチング演算では, 文字列がパターンにマッチするかどうかの判定ができるだけでなく, マッチした文字列を取り出すことができる. これを用いると, 簡単な構文解析が行える.
- 正規表現中のパターンを ( ) で囲っておく. すると, パターンにマッチする文字列が $1 という変数に格納される.

```
my $ip = '192.218.174.172';
if ($ip =~ /^\d+\.\d+\.\d+\.(\d+)/) {
  my $host_number = $1;
}
```

  を実行すると, $host_number には 172 が入る.
- ( ) が複数あると, マッチする文字列が $1, $2, $3, ··· という変数に格納される. これを後方参照と呼ぶ.
- 括弧と変数の対応は「括弧を開いた順」である. すなわち, 正規表現を左から見ていって, 最初に開いた括弧のパターンに一致する文字列が $1 に, 次に開いた括弧のパターンに一致する文字列が $2 に格納される.
- ( ) にマッチする文字列を後方参照したくない ($1, $2 などに格納したくない) 場合は, ( ) ではなく (?:) と書けばよい.

```
my $ip = '192.218.174.172';
if ($ip =~ /^((\d+)\.\d+\.\d+)\.(\d+)/) {
  my $network_number = $1;
  my $first_section = $2;
  my $host_number = $3;
}
```

- 正規表現のマッチングの基本原則は「先頭から最大マッチ」である. マッチする文字列候補が複数ある場合には, まず最初にマッチする文字列が選ばれ, マッチしなくなるところまで文字列が伸ばされる. 例えば,

```
my $ip = ':::abc:def:::ghi::';
$ip =~ /\w+(:\w+)?/;
my $match = $1;
```

  とすると, $match = 'abc:def' となる. マッチングの開始位置は abc となる.
- 最短マッチにしたい場合は, +や*などの量指定子に対して, ?を付けるとよい.
- マッチングにより, 簡単な構文解析を行うことができる.

```
a = y + 12;
```

  のように「右辺に 2 項演算しか現れない代入文」が文字列に与えられたとき, 次のような文によって, 代入先 (a), 演算子 (+), および 2 つのオペランド (y と 12) を切り出すことができる. スペースは複数個あっても無くてもよく, また, 文の前後にスペースが入っていてもよい.

[List 3.2]

```
my $statement = 'a = y + 12;';
if ($statement =~ /^\s*(\w+)\s*=\s*(\w+)\s*(\+|-)\s*(\w+)\s*;\s*$/) {
  my $var = $1;
  my $operation = $3;
  my $operand1 = $2;
  my $operand2 = $4;
  print "$var <- $operation($operand1, $operand2)\n";
}
else {
  print "syntax error\n";
}
```

【EX-3.3】 電子メールアドレス (文字列) を入力し, ユーザ ID (@ より前の部分) とドメイン名 (@より後の部分) を切り出して (スペース区切りで) 表示する, という動作を 3 回繰り返すスクリプト `mextract.pl` を作成せよ. 次のテストケースで動作確認せよ.

文字列	
abc12345@kwansei.ac.jp	abc12345 kwansei.ac.jp
Nagisa.Ishiura@ml.kwansei.ac.jp	Nagisa.Ishiura ml.kwansei.ac.jp
nagisa_ishiura@asip-solutions.com	nagisa_ishiura asip-solutions.com

【EX-3.4】 [List 3.2] の動作を確認し, 乗除算と剰余も扱えるように拡張せよ. さらに, 2 項演算だけでなく `a = y;` という形式の代入文が混在していても処理できる (`a <- y` を出力する) ようにせよ. スクリプト名は `exp.pl` とせよ. 次のテストケースで動作確認せよ.

入力	出力
y = a + 5;	y <- +(a,5)
y = 3 - 4;	y <- -(3,4)
y = s * t;	y <- *(s,t)
y = 8;	y <- 8
y = q / 3;	y <- /(q,3)
y = x;	y <- x
y = 7 % z;	y <- %(7,z)

3.4 置換

- 文字列の置換は,

```
変数 =~ s/正規表現/文字列/
```

という構文で行える. 例えば, 電話番号の '0795-' を '079-5' に置換したければ,

```
my $tel = '0795-65-8300';
$tel =~ s/^0795-/079-5/;
```

と書けば `$a = '079-565-8300'` となる.

- 置換後の文字列には `$1`, `$2` 等の変数も書ける.

```
$tel =~ s/^079(\d)-/079-$1/;
```

と書けば, `\d` にマッチするもの (数字一文字) が `$1` にセットされるので,

```
0795-65-8300 → 079-565-8300
0797-81-1233 → 079-781-1233
```

のような変換が行われる.

- 置換は最初にマッチした文字列 1 つに対してのみ行われる.

```
my $univ = 'Kwansei Gakuin Univ., Ritsumaikan Univ., Doshisha Univ.,';
$univ =~ s/Univ\./University/;
```

と書くと,

```
$univ = 'Kwansei Gakuin University, Ritsumaikan Univ., Doshisha Univ.,';
```

と先頭の Univ. しか置換されない. すべての Univ. を置換したければ g オプションを付けて,

```
$univ =~ s/Univ\./University/g;
```

と書けばよい.

【EX-3.5】 次の操作を 4 回繰り返すスクリプトを作成せよ.

1. co2com.pl

   入力した URL (例えば http://www.yahoo.co.jp のような文字列) の末尾が .co.jp であった場合にはこれを .com に書換え, そうでない場合にはそのまま出力する.

入力	出力
http://www.yahoo.co.jp	http://www.yahoo.com
http://www.kwansei.ac.jp	http://www.kwansei.ac.jp
http://www.co.jp.com	http://www.co.jp.com
http://www.co.jp.co.jp	http://www.co.jp.com

2. tel.pl

   ハイフンの入った電話番号 (例えば 079-565-8300 のような文字列) を入力し, からハイフンを全部消去したもの (0795658300) を出力する.

3.5 記法

- 正規表現はどうしても読みにくくなってしまうので, その対策に x というオプションを用いることがある. このオプションは「パターン中の空白文字を無効化する」というもので, 正規表現に適当なスペーシングを行うことにより, 読みやすくすることができる. スペース記号を含むマッチングは \s を使うしかなくなるので, 若干制約はあるが, スクリプトを読みやすくするために用いるとよい.

```
my $ip = '192.218.174.172';
if ($ip =~ /^  (\d+)  \.  (\d+)  \.  (\d+)  \.  (\d+)  /x) {
  my $host_number = $4;
}
```

- 正規表現の区切りがスラッシュ (/) なので, スラッシュを含む文字列のマッチングの場合 (Unix のファイル名等) には一々バックスラッシュでエスケープする必要があって煩わしいし, 読みにくい. 実は, 正規表現の区切り記号は, スラッシュに限らず**任意の文字** が使える. 例えば,

```
if ($ip =~ /^\/usr\/local\/bin/) {
  $ip =~ s/\/usr\/local\/bin/\/usr\/bin\/;
}
```

のスラッシュの代わりにシャープ (#) を用いた場合は, 次のような記法になる.

```
if ($ip =~ m#^/usr/local/bin#) {
  $ip =~ s#/usr/local/bin#/usr/bin#;
}
```

正規表現の直前の m と s は, それぞれ次に来る文字列がマッチングか置換かを表す. m と s の次に来る任意の一文字が正規表現の区切り記号になる.

【EX-3.6】 【EX-3.4】 の exp.pl 正規表現を x オプションを使って見やすくせよ (exp2.pl として提出せよ).

4 入出力

4.1 簡単なファイル入力の形式

- ファイル入力の最簡形式は次の通り. 最簡だが, 実用ではこの形式が最もよく用いられる.

  [List 4.1]

```
1:  while(<>) {         # これは決まった書き方として覚える
2:    $_ =~ s/^/# /;    # ループを回る毎に $_ に 1 行が読み込まれる
3:    print $_;         # $_ の末尾には改行記号が含まれている
4:  }
```

  – このスクリプトは, コマンド引数で指定されたファイル (指定されない場合には標準入力) から行単位でデータを読み込んで, その先頭に '# ' を付けるという処理を行うものである (行の先頭を '# ' で置換している). このスクリプトを comment.pl とし, sample1.txt を下のような適当なテキストファイルとする.

```
aaa
bbb
ccc
```

  ここで,

```
perl comment.pl sample1.txt
```

  のようにスクリプトを起動すると

```
# aaa
# bbb
# ccc
```

  という出力が得られる.

  – スクリプトの 1 行目は色々なオプションが省略された形式だが, これはこれで決まった書式として覚えておく.

  – ループを回る度に特殊変数 $_ にファイルから次の 1 行が文字列として読み込まれる. この文字列の末尾には改行記号が含まれている.

  – コマンドラインでファイルを複数指定すると, 複数のファイルを指定された順に次々に開いて行単位の読み込みを行う. 例えば, sample2.txt を

```
111
222
333
```

  のような適当なテキストファイルとして,

```
perl comment.pl sample1.txt sample2.txt
```

  とスクリプトを起動すると

```
# aaa
# bbb
# ccc
# 111
# 222
# 333
```

という出力が得られる.

- 逆に, コマンドラインで何もファイルを指定しなければ, 標準入力からの入力となり, キーボードからの入力待ちとなる. (上の例では, 一行入力する毎に出力が行われ, Ctrl+d で終了する). また, Unix や MS-DOS のコマンドパイプに使うこともできる.

- `$_` は省略できる (される).
  - [List 4.1] は次のように書ける.

    [List 4.2]

```
1:  while(<>) {
2:    s/^/# /;     # 置換対象の変数 $_ は省略可能
3:    print;       # 引数を省略すると $_ が仮定される
4:  }
```

  - この他, `split` や `chomp` (文字列の最後に付く改行記号を切り捨てる) 等でも `$_` は省略可能である.

【EX-4.1】 コマンド引数で指定されたテキストファイルに対し, 各行の文字数を表示するスクリプト `llen.pl` を作成せよ. (文字列の長さを返す `length` 関数を用いよ. 行末に改行記号が 1 文字分含まれていることに注意せよ.) 次の内容のテキストファイル

```
1234567
abcdefghijk
Friday, April 13
```

に対して,

```
7
11
16
```

という出力が得られることを確認せよ.

【EX-4.2】 テキストファイルを読み込み, 下記の例のように行番号を付けて出力するスクリプト `lmun.pl` を作成せよ. 行番号の桁数は 2 桁とせよ (`printf` の `"%2d"` を用いればよい.)

```
January          ⇒    1: January
February              2: February
…                     …
December             12: December
```

4.2 例題

- 検索

  数字を含む行を表示させるスクリプトは次の通り.

  [List 4.3]

```
while(<>) {
  if (/\d/) {    # マッチング演算の対象 $_ は省略可能
    print;
  }
}
```

ファイル中の数字列をすべて抜き出して表示するスクリプトは下の通り. 数字列が 1 行に 1 つだけなら (\d+) にマッチングさせるだけでよいが, 複数の数字列があることもあるので, 「行に数字列がある間, 表示して削除する」という処理が必要になる.

[List 4.4]

```
while(<>) {
  while(s/(\d+)//) {  # 数字列があればを $1 にセットし消去
    print "$1\n";
  }
}
```

- 簡単な集計

次のように, 各行に学生の名前と点数が書かれているファイルがあるとする.

```
関学 太郎 : 70 90 80 95 50
理学 次郎 : 40 50 65
情報 司郎 : 90 50 85 45 100 23
```

これを読み込み, 平均点を求めて表示するスクリプトは次のように書ける.

[List 4.5]

```
while(<>) {
  chomp;  # 行末の改行記号を削除
  my ($name, $pointlist) = split(/\s*:\s*/);  # 名前と点数リストを抽出
  $pointlist =~ s/\s*$//;  # 点数リストの末尾にスペースが有れば除去
  my @points = split(/\s+/, $pointlist);  # リストをバラして配列に格納
  my $sum = 0;
  for my $p (@points) {$sum += $p;}  # $sum に合計を求める
  my $n_subjects = scalar @points;  # 科目数
  my $average = $sum / $n_subjects if (0 < $n_subjects);  # 平均
  printf "%-10s :", $name;
  if (defined $average) {printf "%5.1f", $average;}
  else {printf "%5s", '-';}  # 科目数 0 の場合
  print "\n";
}
```

【EX-4.3】 エクセルで次のような学生の得点表を作成し (名前やデータは適当でよい), 「テキスト (タブ区切り)」で別名保存せよ. 各学生に対して, 上位 3 科目の平均点を計算して表示するスクリプト ave.pl を作成せよ. [7]

Kwansei Taro	70	90	80	95	50	
Scitech Jiro		40		50	65	40
Info Saburo	90	50	85	45	100	23

4.3 ファイルを指定した入出力

- 与えられた名前のファイルからデータの入力を行う書式は次の通り. tmp.txt というテキストファイルを読み込み, そのまま出力している.

[7] 日本語を含む場合には, 文字コードの変換が必要になることがある.

[List 4.6]

```
1:  my $filename = 'tmp.txt';
2:  open my $fh, '<', $filename or die "cannot open '$filename'";
3:  while (<$fh>) {
4:    print;
5:  }
6:  close $fh;
```

- 2 行目でファイルをオープンし, 6 行目でクローズしている.
- open の引数は次の通り.
  - 第 1 引数 $fh は「ファイルハンドル」で, これを 3 行目のように < > の中に書くと, 開いたファイルからデータが読み込まれる.
  - 第 2 引数は読み書きのモード.
    - '<' は read,
    - '>' は write,
    - '>>' は append write (最後尾に追記)
  - 第 3 引数はファイル名.
- 2 行目の or 以降は, ファイルのオープンが失敗した時の処理. open は成功すると 1 を, 失敗すると 0 を返す. or は || と同じ論理演算子で, 右辺のは, 左辺が 0 の時のみ評価される. 従って, open が失敗したときのみ die が実行される. die はメッセージを表示して強制終了するものである.

  この「仮定 or die; 」というパターンは, スクリプトのミスのチェックに広く用いる.

  ```
  defined $var or die;
  ```

  だけでも書いておけば, $var がミスで未定義になっているのを早期に発見できる. $var が正数のはずだが念のためにチェックしたい場合は

  ```
  0<$var or die "\$var = $var";
  ```

  と書いておけばよい.

- ファイルへの書き出しは, 次のように書ける.

[List 4.7]

```
1:  my $filename = 'out.txt';
2:  open my $fh, '>', $filename or die "cannot open '$filename'";
3:  print $fh "test\n";
4:  close $fh;
```

- デフォルトのファイルハンドル

  次のファイルハンドルは, 明示的に open することなく使える.

STDOUT	標準出力
STDERR	標準エラー出力
STDIN	標準入力

- print 文でファイルハンドルを指定しないと, STDOUT が省略されていると解釈され, 出力は標準出力に対して行われる.
- エラーメッセージや入力のプロンプトの出力には STDERR を用いる. print STDERR や warn を使うことで, 標準エラー出力に出力することができる.

– 入力の < > は少し複雑である. コマンドラインに引数が指定されていれば, 順次そのファイルを開いて入力する. コマンドライン引数がなければ <STDIN> と等価になる.

- コマンドライン引数の取得

  コマンドライン引数は, 特殊変数 @ARGV から取得できる.

```
perl xxx.pl aaa bbb ccc
```

  や

```
./xxx.pl aaa bbb ccc
```

  でスクリプトを起動すると,

```
@ARGV = ('aaa', 'bbb', 'ccc')
```

  となる. すなわち, i 番目の引数は $ARGV[$i-1] に格納されている. また, 引数の数は scalar @ARGV で取得できる.

  – 2 つの引数 (コピー元とコピー先のファイル名) を受け取ってファイルのコピーを行うスクリプト cpy.pl は次のように書ける.

  [List 4.8]

```
scalar @ARGV == 2 or   # 引数が 2 つあるかどうかチェック
  die "syntax:  $0 FROM_F TO_F\n";  # $0 は起動コマンド名 (ここでは ./cpy.pl)
my $from_f = shift @ARGV;
my $to_f = shift @ARGV;

open my $fin, '<', $from_f or die "cannot open '$from_f'";
open my $fout, '>', $to_f or die "cannot open '$to_f'";
while(<$fin>) { print $fout $_; }
close $fout;
close $fin;
```

【EX-4.4】 テキストファイルの名前 (例えば test.txt) を 1 つ引数として受け取り, その中の英単語の出現回数を数え, 元のファイル名.csv (入力ファイルが test.txt なら test.csv) に書き出すスクリプト fwdist.pl を作成せよ.

- test.txt

```
This is a pen.
That is a cat.
Yes, he is a cat.
```

⇒ test.csv

```
That,1
This,1
Yes,1
a,3
cat,2
he,1
is,3
pen,1
```

4.4 その他

4.4.1 ファイル操作と system 関数

- ファイル操作用に次の関数がある

`unlink FILE1, FILE2, …`	ファイルを消去する
`rename OLDNAME, NEWNAME`	ファイルの名前を変更する
`chmod MOD, FILE1, FILE2, …`	ファイルのモードを変更する

- `system` 関数で, perl の中から OS (主に Unix や Cygwin) のコマンドを呼び出すことができる.

4.4.2 ヒアドキュメント

- 長い文字列やデータをプログラムに埋め込むことができる. `print` 文でよく用いるが, 変数に対する代入も可能である.

```
1:  print << "...";
2:  aaa
3:  bbb
4:  ccc
5:  ddd
6:  ...
```

1 行目の `"..."` の部分は任意の文字列でよい. 次にこの文字列に一致する行 (6 行目) の直前までが一つの文字列として扱われ,

```
aaa
bbb
ccc
ddd
```

と表示される. 文字列への代入は,

```
1:  my $a = << "...";
2:  aaa
3:  bbb
4:  ccc
5:  ddd
6:  ...
```

のように書く.

5　リファレンス

C のポインタに対応するものとして, Perl にはバージョン 5 からリファレンスが導入された.

5.1　スカラのリファレンス

- 基本的な使い方の例は次の通り.
  - C 言語のアドレス演算子 & の代わりに \ を用い, 間接演算子 * の代わりに $ を用いると考えればてっとり早い.

[List 5.1]

```
my $n = 100;
my $p = \$n;  # 変数 $n のリファレンスの取得
print "$$p\n"  # $$p は $n と等価.  100 が出力される.
my $q = $p;
$$q = 30;  # $n = 30 と等価
print "$$p\n"  # 30 が出力される.
```

- 関数呼び出しでの利用
  C 言語と同じように, 関数側で変数の値を書換える必要がある場合は変数のリファレンスを渡す.

[List 5.2]

```
sub double  # 変数のリファレンスを受け取り, 変数の値を 2 倍にする
{
  my ($p) = @_;
  $$p *= 2;
}

my $n = 100;
double(\$n);
print "$n\n";  # 200 と出力される
```

【EX-5.1】 2 つの変数のリファレンスを受け取り, その値を交換する関数 swap を作成し, 動作を確認せよ. 関数 swap とそれをテストするコードをまとめて swap.pl に格納せよ.

5.2　配列のリファレンス

- 基本的な用法は, スカラーのリファレンスの場合とほとんど同じ.

[List 5.3]

```
 1:  my @array = (2, 3, 5, 7, 11, 13);
 2:  my $p = \@array;  # 配列 @array のリファレンスの取得
 3:  print "@$p\n"  # @$p は @array と等価. (2 3 5 7 11 13) が出力される
 4:  my $q = $p;
 5:  push(@$q, 17);  # push(@array,17) と等価
 6:  print "@$p\n"  # (2 3 5 7 11 13 17) が出力される.
 7:
 8:  for my $i (0..$#$p) {
 9:    $$p[$i] = $$p[$i] * 2   # $$p[$i] は $array[$i] と等価;
10:    print "$$p[$i]\n";  # 4, 6, 10, 14, … が出力される
11:  }
```

配列のリファレンスを $p 用いて配列の要素にアクセスする方法は, 9 行目の

```
$$p[$i]
```

だが, これにはもう一つ記法があって,

```
$p->[$i]
```

とも書ける.

※ 非常に良く使うので覚えておくこと.

- 配列の参照渡し

リファレンスを用いると, 関数呼び出しの際に配列全体を渡すという非効率を避けることができる. 2 つの配列を受け取り, それらをベクトルと見たときの内積を返す関数 inner_product は次のように書ける.

[List 5.4]

```
 1:  my @array1 = (2, 3, 4, 5, 6);
 2:  my @array2 = (1, 3, 5, 7, 9);
 3:  my $p = inner_product(\@array1, \@array2);  # リファレンスを渡して呼び出す
 4:  print "$p\n";
 5:
 6:  sub inner_product
 7:  {
 8:    my ($pa, $pb) = @_;  # リファレンスを受け取る
 9:    my $n = scalar @$pa;  # 配列のサイズを $n とする
10:    $n == scalar @$pb || die "ベクトルの次元が違う";  # サイズチェック
11:    my $p = 0;
12:    for (0..$n - 1) {
13:      $p += $pa->[$i] * $pb->[$i];  # 要素の積を累積
14:    }
15:    return $p;
16:  }
```

– 【注意】リファレンスを渡した場合は, 呼び出された側で配列の値を書換えると (当然) 値が書き変わってしまうので注意が必要である. 上の例で, 内積を計算している 12˜ 14 行目のループを

```
12:    while (@$pa) {
13:      my $foo shift @$pa;
13:      my $bar = shift @$pb;
13:      $p += $foo $bar;
14:    }
```

と書くと, 内積は正しく計算されるが, `@array1` と `@array2` は関数呼び出しから戻ると空になってしまう.

- `[ ]` による初期化

  配列を初期化し, そのリファレンスを得るには,

```
my @array = (2, 3, 5, 7);
my $pa = \@array;
```

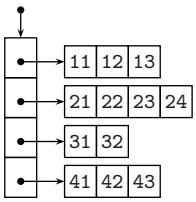
  と書く必要があるが, これは次のように 1 行で書ける.

```
my $pa = [2, 3, 5, 7];
```

  - ※ 非常に便利で良く使うので覚えておくこと.

- リストのリスト

  配列のリファレンスが使えると, 下図のようなリストのリストを作ることができる. Perl には多次元配列がないので, このように配列と配列のリファレンスで処理しなければならない.



  `[ ]` を用いて表現すると, 次のようになる.

```
my $l = [
  [11,12,13],
  [21,22,23,24],
  [31,32],
  [41,42,43]
]
```

  これを表示するスクリプトは次の通り.

```
for my $ll (@$l) {
  for my $e (@$ll) {
    print " $e";
  }
  print "\n";
}
```

  インデックスを用いてアクセスする書き方をすると, 次のようになる.

```
1:  for my $i (0..$#$l) {
2:    for my $j (0..$#{$l->[$i]}) {
3:      print " $l->[$i]->[$j]";
4:    }
5:    print "\n";
}
```

2 行目の下線部のように, 配列のリファレンスを表す式が `$foo` のように単純ではなく `$l->[$i]` のように複雑な場合, 単に `$#` をつけて `$#$l->[$i]` とすると「`($#$l)->[$i]`」と解釈されてエラーが発生するため, 「`$#($l->[$i])`」と解釈させる為に `$#{$l->[$i]}` という書き方をしなければならない.

【EX-5.2】 適当な英文を入力して単語に分解し, 長さが 1 のもの, 長さが 2 のもの, …, にわけて表示するスクリプトを `lenlist.pl` を実装せよ. 下記のリストの関数 `str2lenlist` の本体を埋めてスクリプトを完成させよ.

```
#!/usr/bin/env perl
use strict;
use warnings;

my $s = "This is a pen.  That is a pencil.  I like English.";
my $lenlist = str2lenlist($s);
for my $len (1..$#$lenlist) {
  my $list = defined $lenlist->[$len] ?  join(' ', @{$lenlist->[$len]}) :  '';
  print "$len:  $list\n";
}

sub str2lenlist
{
  # ここを埋めてスクリプトを完成させよ.
}
```

以下の出力が得られることを確認せよ.

```
1: a a I
2: is is
3: pen
4: This That like
5:
6: pencil
7: English
```

5.3 連想配列のリファレンス

- 連想配列のリファレンスも配列と同様

[List 5.5]

```
 1:  my %h = (
 2:    jp => 'Japan',
 3:    kr => 'Korea',
 4:    fr => 'France',
 5:    de => 'Germany',
 6:  );
 7:  my $p = \%h;
 8:
 9:  my @k = keys %$p;       # %$p は %h と等価
10:  print "(@k)\n";          # (jp de fr kr) が出力される
11:
12:  print $$p{jp}, "\n";    # $$p{jp} は $h{jp} と等価.  Japan と表示される
13:  print $p->{jp}, "\n";   # こう書くこともできる.  Japan と表示される
```

- { } による初期化

  [List 5.5] の 1˜ 7 行目の連想配列初期化とリファレンスの取得は, { } を用いて次のように 1 文で書くことができる.

```
 1:  my $p = {
 2:    jp => 'Japan',
 3:    kr => 'Korea',
 4:    fr => 'France',
 5:    de => 'Germany',
 6:  };
```

- 複雑なデータ構造の表現

  連想配列で構造体のエミュレーションができるので, 配列と連想配列とリファレンスを組み合わせると, 複雑なデータ構造が作れる.

  [List 5.6]

```
my $slist = {
  year => 2003,
  school => '理工学部',
  students => [
    { id => '0001', name => '関学 太郎', sex => 'M' },
    { id => '0002', name => '理学 花子', sex => 'F' },
    { id => '0003', name => '情報 次郎', sex => 'M' }
  ]
};
```

  このようなデータ構造に対し, sex が 'M' である学生の id と name を表示するスクリプトは次の通り.

```
for my $st (@{$slist->{students}}) {
  if ($st->sex eq 'M') {
    print "$st->{id} $st->{name}\n";
  }
}
```

- 連想配列の参照渡し

  配列と同様, 関数呼び出しの際に連想配列のリファレンスを渡せば, 連想配列全体が関数に渡されるという問題が解消される.

- ref 関数

  リファレンスが与えられたときに, それが何のリファレンスか (スカラのリファレンスか, 配列のリファレンスか, 連想配列のリファレンスか) を与えるのが ref 関数である.

```
my $foo = \10;
my $bar = [1, 2, 3];
my $baz = {jp => 'Japan', at => 'Austria'};

print ref $foo,"\n";
print ref $bar,"\n";
print ref $baz,"\n";
```

  を実行すると,

```
SCALAR
ARRAY
HASH
```

  が出力される. 関数が引数としてリファレンスを受け取ったときに, その型によって処理を変える場合や, エラー処理に用いることができる.

【EX-5.3】 [List 5.6] で定義されるリファレンス $list を受け取り, 次の形で出力する関数 slist_print を作成せよ. 関数と動作確認用コード ([List 5.6] の末尾に呼び出し slist_print($slist); を追加したもの) を slist.pl に格納せよ.

```
2003 年 理工学部
0001 関学 太郎 (M)
0002 理学 花子 (F)
0003 情報 次郎 (M)
```

【EX-5.4】 リファレンスを一つ受け取り, それがスカラのリファレンス (例えば 5) であれば,

```
SCALAR \5
```

と表示し, 配列 (例えば (1, 2, 3, 5)) のリファレンスであれば,

```
ARRAY [1,2,3,5]
```

と表示し, 連想配列 (例えば ('jp' => 'Japan', 'kr' => 'Korea')) のリファレンスであれば

```
HASH 'jp'=>'Japan', 'kr'=>'Korea'
```

と表示する関数 uprint を作成せよ. 動作確認用のコードと合わせて uprint.pl に格納せよ.