テキスト処理 2009/10/06 出題

高井 実

まえがき

Perl の実行環境を整えて、実際に Perl プログラムに触ってみることが目的。

Windows 環境なら ActivePerl を導入するとよい。

課題内容

1 から 100 までの整数に対して、1 から順番に、3 の倍数のときに three, 5 の倍数のときに five, 15 の倍数のときに threefive, これ以外のときはその数値を出力する Perl プログラムを書く。

出力に関して、ひとつの数字に対する出力ごとに改行を出力する。

Perl プログラムを書く

とりあえず書いてみると次のようなコードが書ける。今回は Windows 環境で ActivePerl を使っているので必要でない shebang line（1 行目の #! で始まる記述）は省略する。

```
1  use strict;
2  use warnings;
3  use diagnostics;
4  for(my $i=1; $i<=100; ++$i){
5    my $ans;
6    if($i % 15 == 0){ $ans = "threefive"; }
7    elsif($i % 3 == 0){ $ans = "three"; }
8    elsif($i % 5 == 0){ $ans = "five"; }
9    else { $ans = $i; }
10   print "$ans¥n";
11 }
```

まず 1〜3 行目について言及しておく。

use strict;	記法を厳密にする．これにより宣言が曖昧な変数への参照などを発見できる．バグに繋がる危険なコードを生まないためにも記述しておくとよい．
use warnings;	警告を出力するようにする．実行時の w オプションでもよい．w オプションの場合は実行ファイルに対してしか警告が出力されない（読み込んでいるパッケージ等には適用されない）．プログラムの目的に合わせて use/no warnings を使うとよい．なお，警告チェックはプログラム実行のパフォーマンスを低下させるため，不要であれば外した方がよい．
use diagnostics;	診断メッセージ（警告の詳細）を出力するようにする．

4〜11 行目がメインとなる部分だが、題意の通りコーディングしている。

このプログラムを実行すると次のような出力が得られる。

1	five	three	13	three	97
2	three	five	14	94	98
three	7	11	threefive	five	three
4	8	three	(16〜92 省略)	three	five

もう少し Perl の構文に慣れてみる

せっかくなので、Perl の構文をもう少し見てみる。

```
1  use strict;
2  use warnings;
3  use diagnostics;
4  $¥ = $/;
5  for(1..100){
6    no strict 'vars';
7    $s = '';
8    $s = 'three' if($_%3 == 0);
9    $s .= 'five' if($_%5 == 0);
10   print $s ? $s : $_;
11 }
```

Perl は変数宣言という概念ないこともあり、ブロックによって変数のスコープが定められるわけではなく、変数にスコープを与えないとグローバル変数となってしまう。このような変数が登場していると strict プラグマが有効な箇所ではそれをエラーとして検出できる。先ほどのコードで $i や $ans を my 関数に渡して、C 言語でいうローカル変数(※)にしていたのはこのためで、一般的な Perl プログラムであれば全ての変数には常に（our/local/my 関数で）スコープを適切に与えるべきである。(※Perl にはローカル変数が 2 種類あるが、my 変数はそのひとつのレキシカル変数である)

しかしながら、局所的にそのような制約を外したいことがある。これを実現するのが "no strict" である。"use strict" は "use strict qw(vars refs subs)" という 3 つの制約をまとめて適用するための記述となっている。一般的な Perl プログラムでは strict プラグマは無効にしたい制約のみを局所的に外すべきである。strict や warnings プラグマはスコープを持つため、"no strict" などによる無効化は宣言したスコープ内に限られる。 7 行目ではスコープを与えずに $s を利用しているが、6 行目で変数に関する制約を無効にしているためこのプログラムはエラーせずに実行することができる。

その他の行についての説明を次に示しておく。

4 行目: $¥ = $/	$¥ は print 関数の行末文字. 末尾に $¥ の値が付加される. $/ は入力時のレコード区切り文字. 通常は改行文字. $¥ に改行文字を代入することで, print 関数は改行を伴うようになる.
5 行目: 1..100	".." は範囲演算子と呼ばれる. ここ(※)では(1, 2, 3, ..., 99, 100)と記述したのと同じ意味になる. (※リストコンテキストと解釈される箇所)
8～10 行目: $_	ループ変数を明示しないループは, 特殊変数 $_ がループ変数となる.
8～9 行目: 後置 if 文	Perl では if 文や for 文などは内容が 1 文であってもブロックが必要であるが, ブロックが不要な後置文にすることができる.
9 行目: $s .= 'five'	5 の倍数であれば $s に 'five' を付加している. 15 の倍数であれば 8 行目で 'three' が代入されるため結果的に $s は 'threefive' となる.
10 行目: $s ? $s : $_	条件演算子 "?:" を用いて, $s が偽であれば $_ を出力している. $_ が 3 か 5 の倍数であれば $s には出力すべき文字列が格納されているが, そうでなければ $s には偽と評価される空文字列が格納されている.

fizzbuzz をショートコーディングしてみる

ソースコードを必要以上に短くすることは決して好ましいとは言えないが、ショートコーディングはその言語に関する理解を深める上では有意義な遊びといえる。

この問題は、Fizz Buzz（フィズ・バズ）という言葉遊びを基にした有名な問題である。

先ほど書き直したコードをもとに考察する。

```
...  .........1.........2.........3.........4.........5.........6.........7.........8.....
 1   print(($_%3?'':'three').($_%5?'':'five')||$_,$/)for 1..100
```

まず、文末のセミコロンは必須ではないので外している。

Perl では関数を呼ぶ際に、print(arg1, arg2); と書かずに print arg1, arg2; と書くことができる。ここで問題なのが、後者のように関数名直後の括弧を省略したつもりで書いたとき、引数リストの 1 個目（arg1 の部分）の先頭に括弧を使ってしまうと前者の構文だと解釈されてしまう点である。ここではひとまず、関数の引数リストの括弧を付けておくことにする。

次に、(print $a || $b) という構文を確認しておく。一般的に、二項演算の論理和(||, or)は、左項が真であれば右項を評価せずに真を返す。左項が偽であれば、右項の真偽値を返す。Perl では真偽値ではなく最後に評価した値そのものを返す。つまり、$a=5 であれば $b の値にかかわらず論理和は 5 を返し、$a=0, $b="hello" であれば "hello" を返す。

print 関数は（'three'.'five' || $_）の形をしており、ループ中のある時点の $_ の値が 3 の倍数でも 5 の倍数でもなければ論理和の左項は空文字列、すなわち偽と評価される値となり $_ が print 関数に渡される。3 の倍数か 5 の倍数であれば、'three', 'five', 'threefive' のいずれかが返される。
さらに、改行を出力するため、カンマで繋いで $/ （デフォルトでは改行文字）を出力している。

この print 関数に対して、最後に 1 から 100 までを繰り返す後置 for 文を書いている。このコード長は 58 バイトである。

無駄な記述を探してもっと短く

条件演算を 2 箇所書いているが、空文字列を返すための '' という記述や、括弧が多いのが気になる。ここで、3 の倍数のときに 'three' を返す箇所は、繰り返し演算子(x)を用いて 'three' x !($_%3) と書けることに気付けば、

```
...  .........1.........2.........3.........4.........5.........6.........7.........8.....
 1   print'three'x!($_%3).'five'x!($_%5)||$_,$/for 1..100
```

と 52 バイトまで短く書くことができる。

さらに、5 の倍数であれば 1 を、そうでなければ 0 を返す剰余の否定 !($_%5) という記述の代わりに、パターンマッチ演算によるマッチング（$_ =~ m/0|5$/）の省略形 /0|5$/ を用いて、末尾が 0 か 5 であるかを調べるようにすると 1 バイト短くすることができる。

```
...  .........1.........2.........3.........4.........5.........6.........7.........8.....
 1   print'three'x!($_%3).'five'x/0|5$/||$_,$/for 1..100
```

これで 51 バイトとなった。

Perl の仕様を利用した 50 バイトコード

さて、次のようなソースコードを書いて実行してみるとどうなるだろうか。

```
1 use warnings;
2 $fruit = 'apple';
3 $data = fruit;
4 print "Answer: $data¥n";
```

3 行目の右辺の fruit の先頭に $ を記述し忘れたように見えるが、これを実行すると、

```
Unquoted string "fruit" may clash with future reserved word at t.pl line 3.
Name "main::fruit" used only once: possible typo at t.pl line 2.
Answer: fruit
```

という 2 行の警告とともに、3 行目に（ソースコード 4 行目の）print 関数の内容が出力された。

注目すべきは 3 行目の出力結果で、$data に 'fruit' という文字列が格納されている点である。これはソースコード 3 行目の（引用符がなく文字列ではない）fruit という記述が、文字列 'fruit' と解釈された結果である。この結果は文脈依存であり常に文字列として解釈されるわけではない。この文脈では fruit を文字列以外（関数など）では解釈できないと判断されたが、例えば、print fruit; のように書くと、fruit は print 関数のファイルハンドラとして解釈され、文字列とは解釈されない。

```
...  .....1.....2.....3.....4.....5.....6.....7.....8.....
 1   print'three'x!($_%3).five x/0|5$/||$_,$/for 1..100
```

'five' の引用符を外して 50 バイトとなった。

限界突破の 49 バイトコード

ここまで来て、もう一度「無駄な記述を探してもっと短く」できないか考えてみる。3 の倍数のときに 'three' を返すために !($_%3) と剰余の結果を反転させているが、剰余の結果である 0 をそのまま使えると都合がよい。5 の倍数に関しても短縮できる可能性があるが、ここで添え字が役に立つ。

```
...  .....1.....2.....3.....4.....5.....6.....7.....8.....
 1   print((three)[$_%3].(five)[$_%5]||$_,$/)for 1..100
```

3 の倍数のときに限り、リストに要素のある[0]がスライスされ 1 つの要素のリスト（'three'）が返される。剰余の結果が 0 以外であればリスト（undef）が返される。この方法でも 50 バイトである。

最後に、print 関数の引数リストの括弧を省略する。括弧を外した際に引数リストの 1 つ目の式が関数の引数リストだと解釈されないように単項演算子のプラス記号(+)を付加する。

```
...  .....1.....2.....3.....4.....5.....6.....7.....8.....
 1   print+(three)[$_%3].(five)[$_%5]||$_,$/for 1..100
```

最終的に 49 バイトコードが実現できた。

本当の最短コード

Perl 5.10 であれば say 関数が使うことでさらに短縮可能だが、ソースコード中にプラグマを書く必要がある。そこで、E オプションをつけた One-Liner を利用することで 45 バイトコードが書ける。

```
...  .....0.....1.....2.....3.....4.....5.....6.....
 1   $ perl -E 'say+(three)[$_%3].(five)[$_%5]||$_ for 1..100'
```

■ 備忘録

Perl の実行環境を整える

まず、perl の実行ができるか確認する。Windows ならコマンドプロンプトを起動して、

```
C:¥>perl -v
```

とでも打ってみる。何らかの Perl がインストールされていれば Perl のバージョン情報が表示される。インストールされていなければ「'perl' は、内部コマンド～として認識されていません」などと表示される。

ActivePerl と検索すると誰かが書いたインストールの手引きがいくつかヒットするので参考にしながらインストール作業を進める。(c.f. http://www.rfs.jp/sb/perl/08/02.html)

http://www.activestate.com/activeperl が ActivePerl の本家ページ。ここから ActivePerl の最新版をダウンロードする。2009 年 10 月現在では 5.10.1 が最新バーションとなっている。

インストールが完了して、環境変数の PATH も通っていれば、前述のコマンドで ActivePerl のバージョンが確認できる。他の Perl が実行されてしまう場合は、使わない Perl を無効にするか、環境変数の PATH の記述順序を変更するなどして ActivePerl が実行されるようにしたほうがよい。

これで環境が整うのだが、Windows の場合、コマンドプロンプトから直接コマンドを打ってもシェルコマンドが使えないことがあるので、Cygwin がインストールされているのであれば、

```
C:¥>bash
```

と打って、bash に入ってシェルコマンドを使えるようにしたほうが操作しやすい。

Perl はスクリプト言語であり、内部ではランタイムコンパイルが行われているがインタプリタで実行されるといえる。プログラムを実行するには、ソースコードを次のように一度 perl に通すだけでよい。

```
$ perl a.pl
```

Perl のソースファイルの拡張子には pl が使われる。

Jcode.pm を入れる

日本語の処理ができるように Jcode.pm を入れておく。PPM 経由では Jcode.pm の最新版が取得できないので CPAN 経由でインストールする。

```
$ perl -MCPAN -e shell
cpan> install Bundle::CPAN
cpan> reload cpan
cpan> install Jcode
cpan> quit
```

cpan に入り、始めに CPAN 自体をアップデートして読み込みし直す。そして Jcode をインストールすればよい。cpan を quit して抜けないと Lockfile が削除されずロックされたままになってしまうことがある。その場合は Perl のインストールディレクトリ（インストール時に Custom setup 画面で変更可能、デフォルトは C:¥Perl）に ″.lock″ ファイルが存在しているはずなので削除すればよい。