第7週の庭いじり

7.1　不思議な循環小数

$\frac{1}{7} = 0.142857\cdots$ は不思議な性質を持っている。正確には、循環節 142857 の性質が面白いのだ。それは

$$\begin{aligned} 142857 \times 2 &= 285714 \\ 142857 \times 3 &= 428571 \\ 142857 \times 4 &= 571428 \\ 142857 \times 5 &= 714285 \\ 142857 \times 6 &= 857142 \end{aligned}$$

となるからだ。なんと、循環節の数字が巡回しているではないか。

こうなってくると、他にも不思議な循環節を持つ分数を調べたくなるだろう。以前、循環節を調べたことを覚えているかい？　庭いじりを始めたばかりの頃は、$\frac{1}{7}$ の循環節を調べるのに苦労したね。余りのリストからひとつずつ値を取り出して、それを計算の種にしながらリストに加えていったんだ。まったく手間のかかる作業だったし、第一、商がどうなっているのか分からずじまいだった。当時はスクリプトを操る力量が不足していたので、仕方なく 1 行ずつの処理を繰り返して目的を達していたのだ。ところが気のせいであったとしても、いまや力量はかなり上昇している。循環小数の循環節を見せてくれるスクリプトを書いてみよう。

対象とする分数は $\frac{1}{n}$ の形で十分だが、n の値によっては困ることがある。循環しないで割り切れてしまう場合があるからだ。割り切れてしまう分数は、必ず $\frac{1}{2^r 5^s}$ の形をしている。したがって、n には 2 と 5 が含まれていないことが条件だ。ただ、この条件の分数だけを対象とすると、$\frac{1}{6}$ のような循環小数も除外されてしまう。しかし $\frac{1}{6}$ は、$\frac{1}{3}$ の循環小数を $\frac{1}{2}$ で分割していると見れば、本質は $\frac{1}{3}$ と同じだ。よって 2 や 5 がひとつでも含まれている分数は、対象から外して構わないだろう。そのほうがスクリプトも簡単にできる。次のスクリプトは、まさに分母に 2 と 5 を含まない分

数の商を表示するものである。

[hs script]

```
quotient :: [Integer] -> [Integer]
quotient (x:xs)
    | r == 1    = drop 1 xs ++ [q]
    | otherwise = quotient (r : xs ++ [q])
    where q = (10*x) `div` head xs
          r = (10*x) `mod` head xs
```

スクリプトは連分数を求めるときに書いたものに似て、少しばかり曲芸的である。それは、たとえば $\frac{1}{7}$ の商を調べる際に引数として 1 と 7 を与えるだろうが、これらは固定したまま商の変化を捕らえたいことが原因である。

また以前のスクリプトでは、$\frac{1}{n}$ の余りはとりあえず $(n-1)$ 回表示させてみた。この方式だと、$\frac{1}{9}$ のような分数では 8 回とも余り 1 が表示されてしまい、無駄な作業であることは否めない。今回は循環する商の表示を目的としているので、$\frac{1}{9}$ は何も `[1,1,1,1,1,1,1,1]` でなく `[1]` となれば十分だ。それに、余分な循環節を表示するようでは、$\frac{1}{113}$ が本当に 112 の循環節を持つ小数かどうかは、綿密に調べなくてはならず効率が悪い。

では、スクリプトが何をしているか見ておこう。引数である $\frac{1}{n}$ は `quotient [1,n]` の形で与える。分子が常に 1 なら、引数を `quotient n` のようにすれば効率的と思うだろうが、あとの処理のために `1,` は必須となっている。ダサダサだね。

ガードの最初の条件は、余りが 1 になった時点がスクリプトの終了であることを示している。分数は $\frac{1}{n}$ で始めているので、余りが 1 であることが最初に戻ることを意味する。余り `r` は `where` 節にある通り、リスト先頭の要素 `x` を 10 倍してから、`xs` の先頭で割った余りである。具体的には、最初の引数は `[1, n]` であるから、$\frac{10}{n}$ を計算していることになる。処理の仕方が少し変わっていると思うだろうが、そうする理由があったのだ。

`otherwise` の行を見てもらいたい。`quotient (r : xs ++ [q])` となっているだろう。つまり `quotient (x:xs)` は常に `quotient (r:xs)` で処理されることになる。このことは、リストの先頭だけが新しい `r` に置き換わることを意味する。逆に言えば `xs` の先頭は変化しない。`xs` の先頭は何だっただろうか。それは $\frac{1}{n}$ の n の値である。要するに、この保証があるために `where` 節における `head xs` はいつでも n の値なのである。

以上のことから、`quotient` は再帰を利用して商を末尾に付け加えることが分かるだろう。そして余りが 1 になったところで `xs` を表示すれば商が列をなして現れる。このとき除数である n は商ではないので、`drop` 関数を用いて取り除いたのである。

実際にスクリプトを実行した例を示しておこう。

(ghci env.)
```
*Main> quotient [1,7]
[1,4,2,8,5,7]
*Main> quotient [1,113]
[0,0,8,8,4,9,5,5,7,5,2,2,1,2,3,8,9,3,8,0,5,3,0,9,7,3,4,5,1,3,2,7,4,3,3,6,2
,8,3,1,8,5,8,4,0,7,0,7,9,6,4,6,0,1,7,6,9,9,1,1,5,0,4,4,2,4,7,7,8,7,6,1,0,6
,1,9,4,6,9,0,2,6,5,4,8,6,7,2,5,6,6,3,7,1,6,8,1,4,1,5,9,2,9,2,0,3,5,3,9,8,2
,3]
```

スクリプトは余りが 1 となったところで処理が終わる。このことは計算を効率的にできた反面、余りが 1 にならない分数は無限ループに陥ってしまう。スクリプトは、はなからそんな分数を引数にとるつもりはないので、注意が必要である。

ところでスクリプトが `[Integer] -> [Integer]` で定義されていたことに気づいただろうか。本当は、ここは `[Int] -> [Int]` で十分である。なぜなら、リストに格納される数は一桁の整数だからだ。しかし、わざわざ無駄なことをしたように見えるのは、商を眺めるだけで終わりではないからである。このあとのためにそうしたのである。

また、`quotient [2,7]` を実行したからといって $\frac{2}{7}$ の商が表示されるわけではない。余りが 1 になって処理が終わるので、循環節を全部表示する前に尻切れになってしまう。そもそも `quotient` 関数は $\frac{1}{n}$ 限定のスクリプトなのだ。分子には 1 しか与えないことが自明なのに、`quotient 1 n` の形で引数をとる。どう考えたって理不尽だよね。だったら最初から、引数には分母しか与えないようにしておけばよい。`quotient` 関数を再定義してみよう。

(ghci env.)
```
*Main> let quotient' n = quotient [1, n]
```

`quotient [1, n]` の形の入力を、`quotient' n` の形に定義し直しただけなのだが、ここには教訓が含まれている。スクリプトを書く際、思い通りのものができなくても、とりあえずきちんと動作するスクリプトが書ければよいということだ。**Haskell** に限らず多くの言語では、関数を組み合わせて別の機能を実現することは普通のことだ。組み合わせ方によっては、不自然な関数だったものが洗練される場合も多い。`quotient` 関数はそんな例だろう。

さて再定義のお陰で、$\frac{1}{n}$ の分母を与えるだけで循環節を表示するようにできた。しかも引数はリストの形でなく、自然な分母の数値を与えればよい。でも、結果はちゃんとリスト表示になる。

(ghci env.)
```
*Main> quotient' 7
[1,4,2,8,5,7]
```

いろいろな観点から、`Int -> [Int]` の型の `quotient` 関数がすぐに作れればよかったのだが、回り道であっても目の前の問題をまず解決することが大事だ。ところで、`quotient` 関数が自然な入力になったのはよいけれど、本当にやりたいことは循環節を定数倍したときの変化を見ることである。そのためには、まだくぐるべき関門があるのだ。