第4週の庭いじり

4.1　ゴールドバッハの予想

4 はちょっとだけ面白い性質を持つ。それは

$$4 = 2 + 2 = 2 \times 2$$

のように同じ数どうしで、和と積に分解できる唯一の自然数だからである。さらに使われている数 2 は、唯一の偶素数でもある。数学は “唯一” という言葉が好きである。

一般に自然数は、**素数**の積に**一意的に**分解できる。一意的とは、掛け算の順番を無視すれば、ただ一通りの積にしかならないことを意味している。しかし自然数は、素数の和に一意的には分解できない。それは

$$10 = 5 + 5 = 3 + 7 = 2 + 3 + 5$$

を見れば分かるように、いくつかの分解の仕方がある。

素因数分解のように一意的に解が求められるものは、分かりやすいし研究の対象にもなりやすい。一方、素数の和にする場合はつかみどころがない。どちらかと言えば面白味に欠ける話題だ、と思ってはいけない。実は大変面白いものがあるのだ。

ゴールドバッハ[1]はオイラー[2]との手紙のやり取りで、“2 以上の偶数は 2 つの素数の和になる” ことを話題にした。当時の 2 人は、1 を素数の仲間に入れていた様子なので、$2 = 1 + 1$ も話題に含まれていたようだ。しかし、われわれは 1 を素数の仲間に入れていないので

4 以上の偶数は 2 つの素数の和になる

と言うべきだろう。現在、この問題に対する**反例**も証明も見つかっていない。

惜しいことに、この中には唯一という単語は含まれない。つまり、4 以上の偶数はただ一組の 2 つの素数の和になる、わけではない。実際 20 は、$20 = 3 + 17 = 7 + 13$ のように 2 組の素数の和で表せる。

[1] クリスティアン・ゴールドバッハ（1690–1764）：ロシアのドイツ人数学者。
[2] レオンハルト・オイラー（1707–1783）：スイスの数学者・物理学者。

ゴールドバッハの予想を検証するには、素数の判定ができなくては困る。そこで素数を判別するスクリプトが必要となる。

[hs script]

```
isprime :: Int -> Bool
isprime n
    | n == 2 = True
    | even n = False
    | odd  n = not (0 `elem` xs)
    where xs = map (n `mod`) [3, 5..n-1]
```

isprime 関数は、引数に与えた数が素数であるかそうでないかを判定するものである。判定というのは、入力してある数に対して、それが素数なら “True” と表示し、素数でなければ “False” と表示するのだ。そして無駄が多いスクリプトでもある。

無駄というのはスクリプトの行数ではなく、処理の仕方である。プログラミングというのは、処理の無駄をなくすために余分なコードを必要とするものだ。isprime 関数はその逆である。余分なコードがないために、無駄な処理をしてしまう。それでも正しい判定はしてくれる。

(ghci env.)

```
*Main> isprime 25
False
*Main> isprime 31
True
*Main> isprime 11111111
False
```

Int -> Bool の型は今までになかったね。これは、整数を引数として真理値—要するに真か偽か—を返す関数である。整数型は Integer じゃなかったっけ？と思ったら、なかなか細かいところに目がいっているね。どちらも整数の型であるが、Int は 64 ビット CPU なら -2^{63} から $+2^{63}$ という範囲を持つ。一方、Integer は範囲無制限だ。今回は格別大きな値を扱わないので、効率重視で Int にしたのだ。

さて、isprime 関数はガードを用いて、2 と偶数に対しては直接素数か合成数を判定している。だから、あとは奇数に対して判定ができればよい。odd n の行を見てほしい。0 `elem` xs というのは、リスト xs に 0 が要素として含まれている場合に True を返す。elem 関数は、以前目にした mod 関数と同じく中置関数として使えるので、ここでは`elem`を用いた。elem A B だと、どっちがどっちの要素なのか分かりにくいんだな。

では、xs には何が格納されているのだろうか。それは、次の where 節を見れば分かる。まず map (n `mod`) [3, 5..n-1] は、n を 3 から n-1 までの奇数で割った余りを求めるという意味で

ある。map は map f [リスト] の形で記述し、関数 f をリストすべての要素に適用して新たなリストを作る関数だ。したがって、たとえば n = 13 に対しては、13 を 3 から 12 までの奇数—すなわち 3 から 11 までの奇数—で割った余りが新たなリストとして xs に格納される。

もう少し分かりやすい例で map の動作を示してみよう。

(ghci env.)
```
Prelude> map (3*) [1, 2, 3, 4, 5]
[3,6,9,12,15]
```

上の例を見れば map がどのような機能を持っているかが分かるはずだ。(3*) という関数をリストの要素ひとつひとつに適用しているのだ。それは

(ghci env.)
```
Prelude> [3*1, 3*2, 3*3, 3*4, 3*5]
[3,6,9,12,15]
```

としているのに等しい。数学的にはベクトル $\vec{a} = (1,\ 2,\ 3,\ 4,\ 5)$ のとき、

$$3\vec{a} = 3(1,\ 2,\ 3,\ 4,\ 5) = (3,\ 6,\ 9,\ 12,\ 15)$$

の計算をしたようなものである。ただし **Haskell** の方がより汎用的である。(3*) の代わりに 13 `mod` としたのが先の例で、

```
(13 `mod`) [3, 5, 7, 9, 11]
    = [13 `mod` 3, 13 `mod` 5, 13 `mod` 7, 13 `mod` 9, 13 `mod` 11]
    = [1, 3, 6, 4, 2]
```

のような調査をしていたのである。この状況は、13 を 3 から 11 までの奇数で割ったとき、いつでも割り切れずに余りが出ていたことを教えてくれる。つまり 13 は素数だ。

もし n が素数でなければ、必ず何らかの奇数で割れているはずだから、xs には少なくともひとつの 0 が要素として存在する。isprime 25 がそのような例で、この場合は (25 `mod`) [3, 5..23] が調査されて xs は [1, 0, 4, ..., 2] になる。`elem`関数は探しているもの—いまは 0 を探している—が要素にあれば True を返すので、0 を含む isprime 25 のリストからは True が返ってしまう。関数名が isprime であるだけに、素数でないときは False を返してほしいのだ。じゃあ、関数名を isnotprime にしちゃおうか。いやいや、それは敗北だ。

False が返ってほしいときに True が返り、True が返ってほしいときに False が返る関数にはどう対処すればよいだろうか。われわれが、True は偽で、False は真であると判断するのはひとつの

方法ではあるが、それは正しい処置ではない。要は真偽を逆に出力してくれればよいわけで、それには `not` という、うってつけの関数がある。それがガードの `odd n` に `not` がある理由である。でも、関数の中に `not` を加えたお陰で、関数名に `not` をつけるという恥ずかしい行為は避けられた。

ところでこのスクリプトはあまり褒められたものではない。特に割り算テストをする際、3, 5, 7, ...,（n の手前の奇数）で割っているが、これらすべてで割るのは時間の無駄なのである。しかし、それ以上にまずいのは引数を `Int` 型で受け取ることだ。`Int` 型は 2^{63} までの数が扱えると言ったね。$2^{10} = 1024 \approx 10^3$ であることを利用すると

$$2^{63} = 2^3 \cdot (2^{10})^6 = 8 \cdot 1024^6 \approx 8 \cdot (10^3)^6 = 8 \cdot 10^{18}$$

であるから、19 桁を超えるような数は扱えないのだ。実際、`isprime` に 19 桁を超える引数を与えるとエラーが返ってくる。

そこで、`isprime` 関数を `Integer -> Bool` で定義し直した `isprime'` を使えば、19 桁を超える引数に対しても結果を返してくれる。実際、11111111111111111111（22 桁）については

(ghci env.)

```
*Main> isprime' 11111111111111111111
False
```

となって、何の問題もない。しかし、うっかり 111111111111111111111（23 桁）を引数にして試すとロクなことにならない。計算効率が悪いせいで画面が凍りつく。ctrl-c で止めよう。結局、`isprime'` 関数は途方もない大きさの数の素数判定には無力なのである。