4.2 素数の表示

isprime 関数は、入力された数が素数かどうか判定するものだった。それなら初めから素数の一覧表があると便利だろう。そこで素数の一覧を表示するスクリプトを書いてみよう。これは与えられた n までの素数をもれなく表示する。

[hs script]
```
primes :: Int -> [Int]
primes 2 = [2]
primes n
    | 0 `elem` xs = primes (n-1)
    | otherwise   = n : primes (n-1)
    where xs = map (n `mod`) (primes (n-1))
```

たとえば、20 までにある素数を表示すると以下のようになるのだが、ちょっと反応が遅く感じることだろう。それに、大きい順に出力されるのも気になるかもしれない。

(ghci env.)
```
*Main> primes 20
[19,17,13,11,7,5,3,2]
```

実際、20 程度なら問題ないが、100 までの素数をこのスクリプトで表示しようとするのはやめておこう。とんでもなく時間がかかるからだ。あとで、ちょっとした修正を施すが、まずスクリプトを説明しよう。

primes はひとつの整数を受け取って、素数のリストを返すので Int -> [Int] という型を持つ。ここでも再帰を使っているので、基底となる素数のリストは [2] である。その上で、n が素数なら既存のリストに追加すればよいし、素数でなければ既存のリストはそのままにすればよい。その観点でスクリプトは書いてある。

n が素数であるかどうかを判断するには、n より小さい素数で割ってみることだ。そうして、どんな素数を持ってきても割り切れないなら、n は新しく見つかった素数ということになる。では、n より小さい素数のリストはどこにあるのだろう。それは $(n-1)$ 以下の素数のリストに他ならない。primes n が n 以下の素数のリストを表すのなら、primes (n-1) に $(n-1)$ 以下の素数のリストがあるはずだ。ならば n は $(n-1)$ 以下の素数のリストで割ってみればよいのだ。それが where 節なのである。

そのとき、(n `mod`) で map されたリスト xs に 0 が含まれてしまったら、primes n のリストは primes (n-1) と同じでよい。しかし、xs に 0 が含まれていなければ、n は新しく見つかった素数なのだから xs に追加すればよい。以上がスクリプトの内容である。

なぜ、これでは実行速度が遅くなるのだろう。それは、3 か所に primes (n-1) が現れているが、そのすべてで素数のリストを構成してしまうからである。そこで、一度構成した素数のリストは使い回すことにすれば、余分な処理がなくなる。primes (n-1) の結果は ps にしまっておこう。

[hs script]

```
primes' :: Int -> [Int]
primes' 2 = [2]
primes' n
    | 0 `elem` xs = ps
    | otherwise   = n : ps
    where ps = primes' (n-1)
          xs = map (n `mod`) ps
```

では、そのように直したスクリプト primes' を実行してみよう。1000 までの素数でも瞬時に表示してくれただろう。数が途中で途切れて表示されるのは、コンピュータの画面の幅によるものだ。

(ghci env.)

```
*Main> primes' 1000
[997,991,983,977,971,967,953,947,941,937,929,919,911,907,887,883,881,877,8
63,859,857,853,839,829,827,823,821,811,809,797,787,773,769,761,757,751,743
,739,733,727,719,709,701,691,683,677,673,661,659,653,647,643,641,631,619,6
17,613,607,601,599,593,587,577,571,569,563,557,547,541,523,521,509,503,499
,491,487,479,467,463,461,457,449,443,439,433,431,421,419,409,401,397,389,3
83,379,373,367,359,353,349,347,337,331,317,313,311,307,293,283,281,277,271
,269,263,257,251,241,239,233,229,227,223,211,199,197,193,191,181,179,173,1
67,163,157,151,149,139,137,131,127,113,109,107,103,101,97,89,83,79,73,71,6
7,61,59,53,47,43,41,37,31,29,23,19,17,13,11,7,5,3,2]
```

ここで、素数の調べ方について話しておこう。素数は 2 を除いてすべて奇数だから、3 以上の奇数の中から、約数を持つ数を順にはねれば素数が残る。スクリプトはエラトステネス[1]の篩（ふるい）と同様の方法で素数を探している。約数を持つということは、既知の素数で割れるということだ。よって、既知の素数で割れる数はバンバンはねてしまえばよい。

まず最初の素数は 2 であるから、primes' 2 —すなわち 2 までの素数のリスト—は [2] となる。スクリプトは n までの素数を求めるものだが、それは n が $n-1$ までの既存の素数で割れるかどうかを調べることが基本になっている。

最後の行を見てほしい。xs = map (n `mod`) ps において、ps は既存の素数リストである。map f [リスト] は、リストの要素に関数 f が一対多対応するんだったね。つまり、n に対して ps の各素数で割った余りが対応し、結果が xs に格納される。n が素数でなければ ps の少なくとも 1 個の素数で割れているはずだから、xs の中に 0 が含まれることになる。それがガードの 1 行目で、

[1] エラトステネス（275B.C.–194B.C.）：ギリシア人の数学・天文学者。

その場合は ps を返せば n までの素数を示したことになる。そうでなければ n は素数なのだから、リストに追加すればよい。

以上のことが再帰で定義されているので、primes' n が実行されると primes' 2 まで遡（さかのぼ）って素数のリストが、その場で作成されてゆく。再び n まで戻ったところで素数一覧の出来上がりだ。

ところで、ここに示した primes' 関数は、効率の観点からはよくないスクリプトである。それは、たとえば 997 が素数かどうか調べるとき、リストにある 991 までの全部の数で割り算をしている。実際は $\sqrt{991} \approx 31.48$ より小さい素数で割れなければ、それより大きな数で割れっこないのだ。また、割り切れてしまえば余りのリストを作ることはない。この差は大きい。スクリプトには改善の余地が残っている。

ところで、本来のエラトステネスの篩のアルゴリズムは次のとおりだ。

a) n を 2 とする

b) n を素数として確定する（○で囲む）

c) n の倍数をすべて消去する

d) 残っている数で、いちばん小さい数を n とする → b) へ戻る

そして、エラトステネスの篩は大変優れたアルゴリズムである。それは、このアルゴリズムで 100 までの素数を調べた一覧を見てもらえば分かる。

	②	③	~~4~~	⑤	~~6~~	⑦	~~8~~	~~9~~	~~10~~
11	~~12~~	13	~~14~~	~~15~~	~~16~~	17	~~18~~	19	~~20~~
~~21~~	~~22~~	23	~~24~~	~~25~~	~~26~~	~~27~~	~~28~~	29	~~30~~
31	~~32~~	~~33~~	~~34~~	~~35~~	~~36~~	37	~~38~~	~~39~~	~~40~~
41	~~42~~	43	~~44~~	~~45~~	~~46~~	47	~~48~~	~~49~~	~~50~~
~~51~~	~~52~~	53	~~54~~	~~55~~	~~56~~	~~57~~	~~58~~	59	~~60~~
61	~~62~~	~~63~~	~~64~~	~~65~~	~~66~~	67	~~68~~	~~69~~	~~70~~
71	~~72~~	73	~~74~~	~~75~~	~~76~~	~~77~~	~~78~~	79	~~80~~
~~81~~	~~82~~	83	~~84~~	~~85~~	~~86~~	~~87~~	~~88~~	89	~~90~~
~~91~~	~~92~~	~~93~~	~~94~~	~~95~~	~~96~~	97	~~98~~	~~99~~	~~100~~

一覧は、7 を素数として確定し、7 の倍数をすべて消したところまでの状況である。次は 11 に○をつけて、11 の倍数を消していく手はずであるが、もうその必要はないのだ。なぜなら、n の倍

数を消すことと、消される運命の数を n で割ることは同じことだからだ。すなわち、100 までの素数を調べるための割り算は $\sqrt{100} = 10$ までよい。したがって、アルゴリズムが 11 に到達したところで、100 までの素数がすべて見つかっているのだ（消えずに残った 25 個が素数）。

ところで、素数が大きい順に表示されることに違和感を覚えるなら `reverse` 関数を使うとよい。リストに `n` を追加するとき `ps++[n]` とすればリストは自然に小さい順に並ぶが、`n : ps` で追加するのが **Haskell** 流だ。この方が処理が速い。

(ghci env.)

```
*Main> reverse (primes' 1000)
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97,10
1,103,107,109,113,127,131,137,139,149,151,157,163,167,173,179,181,191,193,
197,199,211,223,227,229,233,239,241,251,257,263,269,271,277,281,283,293,30
7,311,313,317,331,337,347,349,353,359,367,373,379,383,389,397,401,409,419,
421,431,433,439,443,449,457,461,463,467,479,487,491,499,503,509,521,523,54
1,547,557,563,569,571,577,587,593,599,601,607,613,617,619,631,641,643,647,
653,659,661,673,677,683,691,701,709,719,727,733,739,743,751,757,761,769,77
3,787,797,809,811,821,823,827,829,839,853,857,859,863,877,881,883,887,907,
911,919,929,937,941,947,953,967,971,977,983,991,997]
```

しかし、処理速度が速くなったとはいうものの、スクリプトにはまだ無駄が多い。無駄を減らすヒントは、前回庭いじりをしたときの `isprime` 関数にある。n までの素数を調べるには $\sqrt{n}$ までの調査でよいことから、`isprime` の `where` 節は `[3, 5..sqrt (n-1)]` でよい。ただし、この書き換えだけでは不十分である。なぜなら `sqrt` 関数に整数型の値が与えられてしまうからである。また、リストの要素は整数である必要があるので、`sqrt (n-1)` が返す値は具合が悪い。そこで、`floor (sqrt (fromIntegral (n-1)))` とすればとりあえず OK だが、何とも間抜けな記述である。エレガントなスクリプトは君たちへの宿題としよう。