5.3 フィボナッチ数列

ああ、前々回はひどかった。庭いじりでうっかり蟻の巣を掘り返してしまって、穴から無数の蟻が湧いて出たんだからね。気を取り直して、フィボナッチ数列に戻ろう。

フィボナッチ数列というのは、基本的に $f(n) = f(n-1) + f(n-2)$ の構造を持った数列である。このとき、1, 1 から始めると

$$1,\ 1,\ 2,\ 3,\ 5,\ 8,\ 13,\ 21,\ 34,\ 55,\ 89,\ 144,\ 233,\ 377,\ \ldots$$

が出来上がることは前に見た。では、最初の 2 項に違う数を与えたらどうなるだろう。

そりゃあ **Haskell** が楽にやってくれる、と思ったら残念でした。前に対話モードでフィボナッチ数列を定義したときは、`else` 節に `1` を書いていたと思う。これは初項と 2 項目が 1 だったからだ。では、ここを変えればよいかというと、そうではない。初期値は 2 項分与えなければならない。たとえば 10, 3 で始めたいときは `if`〜`else` では簡単に対処できないのだ。

だったらスクリプトを書けばいいんだ。これでどうだろう。

[hs script]

```
fibonacci :: Int -> [Int]
fibonacci 1 = [3, 10]
fibonacci n = head (fibonacci (n-1)) + head (tail (fibonacci (n-1))) :
              fibonacci (n-1)
```

`fibonacci n` の行は 2 行に分割されているが、1 行で入力してほしい。それにしても見苦しい式だ。`head (fibonacci (n-1))` はまだよいとしても、`head (tail (fibonacci (n-1)))` はなんだかなあ、という感じである。しかし、実行すると

(ghci env.)

```
*Main> fibonacci 10
[505,312,193,119,74,45,29,16,13,3,10]
```

のように第 10 項までのフィボナッチ数を表示する。初項は第 0 項と数えていることに注意して。そして右から左へ見るように。これでちゃんと機能するのは、**Haskell** がリストの分け方を

$$[\overbrace{a_{n-1}}^{\texttt{head}},\ \overbrace{a_{n-2},\ a_{n-3},\ \ldots,\ a_1,\ a_0}^{\texttt{tail}}]$$

のようにしていて、`head` はリストの a_{n-1} を、`tail` に対する `head` はリストの a_{n-2} を指すからである。要するに `head` はリストの先頭の要素を指し、`tail` は残りすべての要素を指す。ちなみに図は `fibonacci (n-1)` の様子を模していて、この先頭に a_n が付け加わったリストが `fibonacci n` である。

ところで数列は逆順に出力されるが、これは `reverse (fibonacci 10)` と書けば第 0 項から並ぶので大した問題ではない。問題は初項が `[3, 10]` に固定されてしまっていることだ。念のために言うと、スクリプトの仕様が逆順で表示されるということなので、$a_0 = 10$、$a_1 = 3$ であることに注意しよう。どうも注意が多いね。できれば、初項を引数で受け取って調べたいのだ。その方が、いちいちスクリプトを書き換えなくて済むからね。で、今度はこうしてみた。

[hs script]

```
fibonacci' :: [Int] -> [Int]
fibonacci' (x:xs)
    | length xs == last xs = init (x:xs)
    | otherwise            = fibonacci' ((x + head xs) : x : xs)
```

これは、最初の 2 項を引数にしてフィボナッチ数列を表示するものだ。しかし、最初の 2 項だけでは際限なく数列を表示しかねないので、必要な項数も引数で与えることにした。さっきと同じ出力になっているはずだ。

(ghci env.)

```
*Main> fibonacci' [3, 10, 11]
[505,312,193,119,74,45,29,16,13,3,10]
```

これでも同じように機能するのは、**Haskell** のもうひとつのリストの分け方が

$$[\underbrace{a_{n-1},\ a_{n-2},\ a_{n-3},\ \cdots,\ a_0}_{\texttt{init}},\ \underbrace{n}_{\texttt{last}}]$$

であるためである。要するに `last` はリストの末尾の要素を指し、`init` は残りすべての要素を指す。図は `fibonacci'` 関数の引数の構造を示している。

これなら自由に初項を与えられるのだが、`fibonacci` 関数の引数の 3 番目は第 n 項を意味するのではなく項数であることに注意しよう。一貫性に欠けることおびただしい。で、スクリプトはだいぶ妙なことをしてしまった。

このスクリプトは、フィボナッチ数列を漸化式としてではなく、手続きとしてとらえている。多くのプログラミング言語が得意とするものだが、**Haskell** にとってはそうではない。それでもスクリプトは 4 行でしかない。手続きと言うのは、リストの先頭と次の要素の和をリストに付け加えるというものである。それは `(x + head xs) : x : xs` で表現できる。

手続き型の言語では計算回数の上限を与える書き方をするが、**Haskell** では違う。それでも必要な項数を知りたければ `length` 関数が使える。リストの末尾には項数が格納されるようになっていたはずだから、リストの長さがその値に等しければよいことになる。そのことを表すガードの表現は、単に `length xs == last xs = (x:xs)` でよかったけれど、リスト末尾はフィボナッチ数列

とは関係ない数値であるから取り除いてよい。リストの最後の要素を取り除くということは、リストの末尾以外を取り出せばよいので init 関数が適している。

さて、これで自由な初期値から様々なフィボナッチ数列が生成される運びとなったが、実は重要な問題が潜んでいるのだ。たとえば fibonacci' [1, 1, 93] は

(ghci env.)

```
*Main> fibonacci' [1, 1, 93]
[-6246583658587674878,7540113804746346429,4660046610375530309,288006719437
0816120,1779979416004714189,1100087778366101931,679891637638612258,4201961
40727489673, ... （途中省略） ...
... ,6765,4181,2584,1597,987,610,377,233,144,89,55,34,21,13,8,5,3,2,1,1]
```

となってしまう[1]。先頭の-6246583658587674878って何？　フィボナッチ数列に負の値が現れる原因は、関数の型に Int を使っているからだ。この不具合は前に述べたね。Integer 型にすればよいだろうと思うかもしれないが、リストの中に項数を含める仕様のままでは感心しない。もっとも Integer 型では length xs の型が合わない。実際:t コマンドで調べると length 関数は Int を返す関数であることが分かる。

元祖である finonacci 関数はどうだろう。この関数なら Integer にすることに異論はない。しかし、これも 3 か所にわたって fibonacci (n-1) を使っているため、実行速度がひどく悪い。間違っても fibonacci 93 なんて試しちゃだめだよ。それなら、以前 primes 関数で対処したように fibonacci (n-1) を置き換える方法でしのげばよいだろうと思うかもしれないが、初期値が固定されている関数では使い勝手が悪い。やっぱり、別の手を考えなくちゃ。項数はリストに含めないで渡したいんだ。

そういうことなら、リストから必要な数の要素を取り出す take という関数があったはずだ。たとえば take 10 [1..] と書けば、[1..] が 1 から始まる数の無限リストであっても、先頭から 10 項だけ取り出してくれて [1,2,3,4,5,6,7,8,9,10] が返ってくる。どう考えても、この方式がすっきりしているじゃないか。それで、take 関数のような使い方になるように、fibonacci' 関数を書き直してみた。

[hs script]

```
fibtake :: Int -> [Integer] -> [Integer]
fibtake n (x:xs)
    | length xs == n = reverse (x:xs)
    | otherwise      = fibtake n ((x + head xs) : x : xs)
```

これは fibtake 92 [1, 1] のように書いて、1, 1 から始まるフィボナッチ数列を初項から第 92

[1]機器の CPU によっては、これと異なる値になるかもしれない。

項まで出力する。初項は第 0 項と数えているから、つまり 93 項分が出力される。そして、ここには変な技巧はない。大きなフィボナッチ数に対応できるよう、整数の型は Integer にしてある。この場合は比較が length xs == n であるため不適合は起こらない。ついでに出力が自然になるように、あらかじめ reverse 関数を組み込んでおいた。実際に fibtake 92 [1, 1] を試してみたのがこれだ。第 93 項は正しい正の値になっているね。

(ghci env.)

```
*Main> fibtake 92 [1, 1]
[1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,1597,2584,4181,6765,10946,
17711,28657,46368,75025,121393,196418,317811,514229,832040,1346269,2178309
,3524578, ... （途中省略） ...
... ,4660046610375530309,7540113804746346429,12200160415121876738]
```

この関数は fibtake 1000 [1, 1] 程度なら軽く処理してくれる。1000 項目のフィボナッチ数なんて興味ないかもしれないけれど、フィボナッチ数がどれほど簡単に大きな数になるのかを実感するにはよい。でも、もう一度際限なく湧き出る蟻を見ることになるんだけど。

ところで、fibonacci' [1, 1, 93] では 93 項分（すなわち第 92 項まで）の数列が出力され、fibtake 92 [1, 1] では第 92 項まで（すなわち 93 項分）の数列が出力されるのは仕様として明らかに変だ。fibonacci' [1, 1, 93] は暫定的とは言え、数列のリストに項数を含ませるのは紛らわしいことこの上ない。fibtake 92 [1, 1] に至っては、take 92 と言っておきながら 93 項分取り出すのは如何（いかが）なものか。君たちにはこれを「他山（たざん）の石」としてもらえれば幸いだ。

しかし、それよりもスクリプト自体が直感的じゃないよね。発端の妙なスクリプトを色々いじってきたからだが、直感的には

[hs script]

```
fibs :: Int -> [Integer]
fibs 1 = [1]
fibs 2 = [1, 1]
fibs n = fibs (n-1) ++ [last (fibs (n-1)) + last (fibs (n-2))]
```

なのだろう。でも、これだと再帰のせいで実行速度が遅いのだ。しかも、1, 1, ... 始まりに固定される。自由度があって軽快で分かりやすいスクリプトに仕立てるのは、君たちに任せることにしよう。