5.4 2進数

`Int` と `Integer` の振る舞いの違いに翻弄されたかもしれない。ここで 2 進数に触れておくのも悪くないだろう。

まず、2 進数について理解を深めよう。10 進数も 2 進数も**位取り**をして、いくらでも大きな数や小さな数を表現している。ただ、位取りの基準が違うのである。10 進数の位取りの基本は 10 である。当たり前か。そして、当たり前の延長で、2 進数の位取りの基本は 2 である。これは当たり前じゃないかな？　では、次の表を見てもらいたい。

10 進数の位	$\cdots$	$\frac{1}{10^4}$	$\frac{1}{10^3}$	$\frac{1}{10^2}$	$\frac{1}{10}$	1	10	10^2	10^3	10^4	$\cdots$
2 進数の位	$\cdots$	$\frac{1}{2^4}$	$\frac{1}{2^3}$	$\frac{1}{2^2}$	$\frac{1}{2}$	1	2	2^2	2^3	2^4	$\cdots$

たとえば 10 進数の 251 は、10^2 の位、10 の位、1 の位にそれぞれ 2、5、1 が使われているので

$$2 \times 10^2 + 5 \times 10 + 1 \times 1 = 251$$

なのである。同じことは 2 進数の 11111011 にも言える。この数は、2^7 の位、2^6 の位、2^5 の位、2^4 の位、2^3 の位、2^2 の位、2 の位、1 の位にそれぞれ 1、1、1、1、1、0、1、1 が使われているので

$$1 \times 2^7 + 1 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2 + 1 \times 1$$

なのである。さあ、実際に計算してみよう。251 になっただろう。

同様に、小数も 2 進数で表記できる。たとえば 2 進数の 1001.011 を 10 進数にするには次のようにする。

$$1 \times 2^3 + 0 \times 2^2 + 0 \times 2 + 1 \times 1 + 0 \times \frac{1}{2} + 1 \times \frac{1}{2^2} + 1 \times \frac{1}{2^3} = 9.375$$

へえ、1001.011 は 9.375 なんだ。いずれも有限小数である。でも、2 進数と 10 進数の関係は、そう単純な話ではない。

たとえば、10 進数の 0.1 を例にとろう。もちろん有限小数で、0.1 の位に 1 がある数である。2 進数ではどう表せるだろうか。2 進数で $0.abcde\cdots$ で表される数は、a は 0.5 の位、b は 0.25 の位、c は 0.125 の位、d は 0.0625 の位、e は 0.03125 の位、... である。例に示した 10 進数は 0.1 だから、0.5 の位、0.25 の位、0.125 の位は、桁の数が 0 のはずだ。もし、これらの桁に 1 があれば、その数は 0.1 より大きくなってしまうからだ。0.0625 の位には 1 がある。0.03125 の位にも 1 がある。

これで合計 0.09375 になった。次は 0.015625 の位だが、この桁は 0 である。1 であれば合計が 0.1 を超えてしまうからだ。

さて、このようなことを続けていけば分かるが、この操作でちょうど 0.1 にすることはできない。これ以上詳しく調べないけれど、10 進数の 0.1 は 2 進数では 0.000110011$\cdots$ となる。つまり、無限循環小数なのだ。要するに小数を扱う限り、10 進数と 2 進数はぴったり同じ値を扱っているわけではないのである。だいぶ前に、**Haskell** が $1+\frac{2}{3}$ の値を、末尾が微妙に違う小数で表したことを覚えているだろうか。2 進数と 10 進数の違いから誤差が入り込むのは仕方ないことなのだ。

さて、**Haskell** が少し変な値を表示するのは小数に限らない。フィボナッチ数列を計算したとき、大きなフィボナッチ数が負の値になってしまう現象も、**Haskell** に限らず多くのプログラミング言語に特有のものだ。ちょっと実験してみよう。

(ghci env.)
```
*Main> maxBound::Int
9223372036854775807
*Main> minBound::Int
-9223372036854775808
```

`maxBound` と `minBound` は上限と下限を教えてくれる関数である。`Int` 型の整数を調べた結果だが、64 ビット CPU なのでこの数値である。32 ビット CPU なら上限は `2147483647`、下限は `-2147483648` となる。ところが、上限と言っておきながら上限を超えることもある。

(ghci env.)
```
*Main> 9223372036854775807+1
9223372036854775808
```

それは **Haskell** が、とくに指定しなければ整数の計算を `Integer` 型で行うからである。`Integer` 型なら上限はない。`Int` 型に上限と下限があることは次の計算で分かる。

(ghci env.)
```
*Main> (9223372036854775807+1)::Int
-9223372036854775808
*Main> (-9223372036854775808-1)::Int
9223372036854775807
```

なんと、`1` を足したら負の値に、`1` を引いたら正の値になってしまった。理由は **Haskell** が整数を**符号付き 2 進数**で扱ってるからだ。実際に **Haskell** が扱っている 2^{64} の大きさの `Int` 型で考えるのは大変なので、2^4 程度で説明しよう。2^4 のサイズの数は 0000〜1111 まで、すなわち 0〜15 までが扱える。しかし、これでは正の数しか扱えないので、負の数を扱いたいときは最高位が 1 である数を負の数として扱う。つまり、0000〜0111 までが正の数、1000〜1111 までが負の数である。

ここで注意が要るのだが、負の数の最高位 1 が負の符号、すなわち “–” の意味で使われていないということである。

2 進数	0000	0001	0010	$\cdots$	0111	1000	1001	1010	$\cdots$	1111
10 進数?	0	1	2	$\cdots$	7	-0?	-1?	-2?	$\cdots$	-7?

もし最高位を “-” 扱いすると、正の数の対比から $1001 \to$ -001 $\to -1$ というのは理解できても $1000 \to$ -000 $\to -0$ は受け入れられないだろう。それに、2 進数の計算規則では $1001 + 1 = 1010$ であるものが、10 進数に書き直したとき $(-1) + 1 = -2$ ではおかしいだろう。そこで、2^4 のサイズならば

2 進数	0000	0001	0010	$\cdots$	0111	1000	1001	1010	$\cdots$	1111
10 進数	0	1	2	$\cdots$	7	-8	-7	-6	$\cdots$	-1

という対応になる。1000 が -8 であることに注意しよう。1000 から 1111 までは 8 個の数があるんだからね。これなら、$1001 + 1 = 1010$ は $(-8) + 1 = -7$ となって、何ら矛盾は起こらない。この例は 4 桁なのだから、$1111 + 1 = 10000$ は最高位の 1 は捨てられて $1111 + 1 = 0000$ である。これも $(-1) + 1 = 0$ であるから問題ない。

唯一の齟齬（そご）が $0111 + 1 = 1000$、すなわち $7 + 1 = -8$ なのである。そして `Int` 型では、唯一の齟齬が $9223372036854775807 + 1 = -9223372036854775808$ で起こったのである。**Haskell** にしてみれば

$$\begin{aligned}&0111111111111111111111111111111111111111111111111111111111111111 + 1\\&\quad = 1000000000000000000000000000000000000000000000000000000000000000\end{aligned}$$

を計算しただけである。それにしても **Haskell**って、こんな桁数の計算を平然と行っていたんだ。なんて、賢いんだろう。

ちなみに、下限から 1 を引いた $-9223372036854775808 - 1 = 9223372036854775807$ の計算は、上限の計算式において $+1$ を移項したものであることに注意しよう。当然それは、2 進数の計算においても移項したものであるから、

$$\begin{aligned}&1000000000000000000000000000000000000000000000000000000000000000 - 1\\&\quad = 0111111111111111111111111111111111111111111111111111111111111111\end{aligned}$$

ということである。ちゃんと辻褄（つじつま）が合っているじゃないか。

でも、辻褄が合うからといって型に無頓着（むとんちゃく）ではいけない。自分が書いたスクリプトが Int 型で動作していることに気づかずにいると足元をすくわれる。単純なスクリプトで確認しておこう。

[hs script]

```
overint :: Int -> Int
overint n = 2^63 + n
```

overint 関数は意図的にオーバーフローを起こす。

(ghci env.)

```
*Main> overint 1
-9223372036854775807
```

Haskell は目ざとくエラーを発見してくれるが、この場合は正常な動作と認めている。過信は禁物なのだ。