第2週の庭いじり

2.1 関数

数学に**関数**は付きものである。関数は特別難しいものではない。記述の仕方に慣れさえすれば、関数はむしろ便利で使いやすいのである。それに **Haskell** は純粋関数型プログラミング言語と呼ばれているわけだから、関数から逃れることはできない。でも、もうしばらくは対話モードで進めよう。

さて、関数 $f(x) = x^2$ があるとする。これは、f という名の関数に (何がし)^2 の機能を持たせている。そのため関数 f に 5 を与えると、25 という値が返ってくるのだ。また関数 $g(x) = 2^x$ は、g という名の関数に $2^{(\text{何がし})}$ の機能を持たせている。よって関数 g に 10 を与えると、1024 という値が返ってくるのだ。

ここでちょっと変な感じを持った人がいるだろうか？　私が関数 f, g と言ったことに対してだ。だが、これでよい。$f(x)$ と書くのは、関数 f が変数 x を使うことを明らかにするためなのだ。もし皆が皆、変数には x しか使わないと取り決めていたら、$f = x^2$, $g = 2^x$ で十分である。しかし、変数には x 以外の文字を使うことはよくあるし、変数が 2 つ以上ある関数だってある。さらには文字定数を利用する場合があることを思えば、関数名に続けて ( ) を付け加えることが不可欠なのである。

関数の例をあと 2 つ提示しよう。

最初は $f(x,\ y) = x^2 - 3y$ でどうだろう。関数 f は変数に x, y を使う。その結果返ってくるのが $x^2 - 3y$ で計算された値だ。だから $f(5,\ 10)$ と書けば、$x = 5$, $y = 10$ を代入して計算した値 -5 が返ってくることになる。一般に数学で使う関数は、実数値を受け取って実数値を返すので、$f(0.8,\ 0.1)$ でも計算可能で 0.34 が返ってくる。

2 つ目の関数は $g(x) = \lfloor x \rfloor$ でどうだろう。見慣れない記号 $\lfloor\ \rfloor$ はフロア記号と呼ばれる。フロア記号を簡単に説明すると、与えられた値を超えない整数のうち、最大のものを返す機能を持っている。むむ、難しい表現だ。具体的には $g(3.14)$ と書けば 3 が返され、$g(-3.14)$ と書けば -4 が返されるのだ。いわゆる “切り捨て” である。フロア（floor）の名称から想像できるように、これは建

物の “床” を意味する。3.14 階という表現はおかしいが、3 階の部屋の中空に 3.14 階があると思えば、その床は 3 階だから $\lfloor 3.14 \rfloor = 3$ とみなすのである。

フロアと対（つい）になる関数はシーリング（ceiling）と呼ばれ、$\lceil x \rceil$ で表される。$\lceil x \rceil$ は、x を下回らない最小の整数である。具体的には $\lceil 3.14 \rceil = 4$、$\lceil -3.14 \rceil = -3$ などである。**Haskell** で利用するときは、`floor 3.14` や `ceiling (-3.14)` でよい。ただし、負の数には必ず ( ) を付けること。付け忘れると **Haskell** に文句を言われるのでね。

いま 2 つの関数を例に出してみたが、値を求めるためにする記述は、いずれも $f(5,\ 10)$ や $g(3.14)$ のように簡便だ。にもかかわらず -5 や 3 の値が返ってくるのは、裏方で f や g がせっせと $5^2 - 3 \times 10$ や $\lfloor 3.14 \rfloor$ の計算をしているからである。また、$f(3,\ 8)$ のように別の値を与えれば、それに応じた値が返ってくる。これは、関数がする仕事がきちんと決められているからである。

何のことはない。関数とは、与えられた値を別の値に加工する手続きなのだ。

ここでは**指数関数**を話題に取り上げる。それも、もっとも基本的な $f(x) = 2^x$ にしておこう。しかしその前に、**Haskell** における関数の仕組みを知るために、x の値を入力すると 2^x の値を表示するスクリプトを書いておこう。

(ghci env.)

```
Prelude> let f x = 2^x
Prelude> f 5
32
```

Haskell の対話モードでは、代入だけでなく関数の定義も `let` で行う（`let` は省略してもよい）。いま定義したいのは $f(x) = 2^x$ なので、`let f x = 2^x` とする。( ) を使っていないが、“まんま” の定義である。当然 `f 5` で 32 が返ってくる。簡単すぎて拍子抜けしたのではないかな？

じゃあ、$2^{1.23}$ はどうだろう。

(ghci env.)

```
Prelude> f 1.23

<interactive>:23:1: error:
    • Could not deduce (Integral b0) arising from a use of ‘f’
      from the context: Num a
        bound by the inferred type of it :: Num a => a
        at <interactive>:23:1-6
    :（以下省略）
```

これは参った。何を言っているか分からないね。でも、純粋な数学の関数と同じ振る舞いはできないってことだ。まあ、庭をいじり始めたばかりだから、いまは **Haskell** の機嫌を損ねるようなことはやめておこう。そのためには、機嫌を損ねた理由を知っておかなくちゃ。

なぜ、**Haskell** が癇癪（かんしゃく）を起こしたか説明しよう。そこで、対話モードにおいて`:t f`と入力してほしい。

(ghci env.)
```
Prelude> :t f
f :: (Integral b, Num a) => b -> a
```

何やら意味不明なものが表示されたと思うかもしれない。`:t` は引数に与えた式や関数の**型**を教えてくれる命令である。よって`:t f` を打ち込むと、いま定義してある関数 `f` の型を表示するわけだ。じゃあ、`f :: (Integral b, Num a) => b -> a` って何？

`f ::`は「関数 `f` の型は：」と読めばよいだろう。で、その後ろは「`(Integral b, Num a)` 型クラスに属していて、`b` を引数にとり、`a` を返す」と読んでみよう。ますます分からないね。でも $f(x) = 2^x$ が、`b` を引数にとり `a` を返す関数であるということは、**Haskell** 的には $f(x) = 2^x$ は `a = 2^b` と見ていることになるだろう。その上で、`Num` は実数を、`Integral` は整数のみを表す型クラスであると聞けば、さっきのエラーも合点がいくだろう。つまり、`f x = 2^x` は整数引数のみを受け付けると定義されていたのである。

定義だって？　われわれが定義したのは `f x = 2^x` という関数だったはずだ。いつの間に型まで定義されたのだろう？　それに、2^x が実数を返すという定義は、少し変じゃないかと思うかもしれないね。x が整数なら関数の値も整数に限られるんだから。でも、こういうことである。たとえば $f(x) = 2.5^x$ は定義できるのだ。実際、

(ghci env.)
```
Prelude> let f x = (2.5)^x
Prelude> f 2
6.25
```

のようになる。そこで、このように定義された関数の型を調べると

(ghci env.)
```
Prelude> :t f
f :: (Integral b, Fractional a) => b -> a
```

が表示される。おや？　`Fractional`って何？　さっきと違う定義だね。実は **Haskell** は型推論をしているのだ。いまは `(2.5)^x` を定義したので、**Haskell** は `2.5` を見て `Fractional`、つまり有理数を返せばよいと判断している。でも、相変わらず引数は `Integral b` だから整数しか使えないことに変わりない。**Haskell** では基本的に、指数に使われる値は整数値なのである。

これまでに何度か見てきたが、**Haskell** は型の制約が大きい。それなら **Haskell** は、実数を引数にできないのだろうか。いや、そんなことはない。

(ghci env.)

```
Prelude> let f x = sqrt x
Prelude> f 5
2.23606797749979
Prelude> f 1.23
1.1090536506409416
Prelude> :t f
f :: Floating a => a -> a
```

$f(x) = \sqrt{x}$ を定義して、引数に 5 と 1.23 を与えた例を示してみた。見ての通り関数の型は `Floating a => a -> a` であるから、浮動小数点数の型クラスに属し、浮動小数点数を引数にとり、浮動小数点数を返す関数である。本当は、解釈が正しいとは言えないのだが、まあ、いまのところはこの程度の理解で済ませておこう。