2.2 再帰的に定義する関数

Haskell でいくつかの関数を定義して実際の動作を調べたが、**Haskell** が得意とする関数は**再帰**的に定義できる関数である。再帰って何だろう。“お約束” の例を示して説明しよう。再帰の例で必ず登場するのが $n!$ の計算である。$n!$ は「n の階乗（かいじょう）」と読み、

$$n! = n(n-1)(n-2)(n-3)\cdots\cdots 3\cdot 2\cdot 1$$

と定義される関数で、たとえば $5! = 5\cdot 4\cdot 3\cdot 2\cdot 1 = 120$ と計算する。この計算はすぐにびっくりするような大きな数になってしまうので、記号 “!” を使うのだと聞いたことがあるが本当かどうかは知らない。ちなみに、諸般の事情で $0! = 1$ と定義されている。

諸般の事情のひとつを説明しておこう。たとえば n 個あるものから r 個のものを取り出すとき、何通りの取り出し方があるか、という問題を考える。庭いじりの最中にものを数えても仕方ないので、結論だけ言っておこう。この場合は $\dfrac{n!}{r!(n-r)!}$ 通りの取り出し方があることが知られている。具体的には、5 個のもの（a, b, c, d, e）から 2 個を取り出す方法は

$$(a,\ b),\ (a,\ c),\ (a,\ d),\ (a,\ e),\ (b,\ c),\ (b,\ d),\ (b,\ e),\ (c,\ d),\ (c,\ e),\ (d,\ e)$$

の 10 通りであり、これは $\dfrac{5!}{2!(5-2)!} = \dfrac{5\cdot 4\cdot 3\cdot 2\cdot 1}{2\cdot 1\cdot 3\cdot 2\cdot 1} = 10$ の計算結果に等しい。

では、5 個のものから 5 個を取り出す方法は何通りか。もちろん全部取り出すという 1 通りの方法しかない。これを計算で求めようとすると $\dfrac{5!}{5!(5-5)!} = \dfrac{5!}{5!0!} = \dfrac{1}{0!}$ となるが、これは 1 に等しいはずだから、$0! = 1$ でないと具合が悪い。つまり、つじつまが合うようにするために $0! = 1$ と定義せざるを得なかったのである。

さて、この定義に従うと、文字を $(n-1)!$ のように使えば $(n-1)! = (n-1)(n-2)\cdots\cdots 1$ の計算をすることになる。つまり $n! = n\times(n-1)!$ である。ここ、ポイントだよ。言い換えれば $n!$ の計算は $(n-1)!$—すなわち n のひとつ前の階乗—に n を掛けたものになっている。そして、これが「再帰」ということなのだ。平たく言えば、

階乗の計算をするためには、（ひとつまえの状態の）階乗の計算を使えばよい

ということである。鶏と卵の関係めいているが、上の表現はまさに再帰の本質を突いているはずだ。

このことから、$f(n) = n!$ のとき、この関数のひとつ前の状態は $f(n-1) = (n-1)!$ と書ける。$n!$ の計算は、ひとつ前の階乗の計算—それは $(n-1)!$ だ！—に n を掛ければよいのだから、$f(n)$ と $f(n-1)$ を用いて表すと $f(n) = n\times f(n-1)$ である。そして、これは階乗の定義になっている。**Haskell** 流に行うと以下のようになる。

(ghci env.)
```
Prelude> let f n = if n > 0 then n * f (n-1) else 1
Prelude> f 4
24
Prelude> f 10
3628800
```

もっとも、このスクリプトは対話モードで実行しているため、**Haskell** 流とは名ばかりで、だいぶ野暮ったいものになっている。本当の **Haskell** 流スクリプトはあとで示すことにして、まずスクリプトの説明をしておこう。

階乗の計算式は `f n =` に続く部分で、それは `if` 式で定義されている。他のプログラミング言語では `if`“文” と言うかもしれないが、**Haskell** では “式”、つまり関数として扱っている。ただ、細かいことを気にしなければ、文も式も大差ない。要するに、要求された値に対して適切な処理がなされるだけだから。

スクリプトは階乗の定義 $f(n) = n \times f(n-1)$ をそのまま使っている。ただし、それが有効なのは $n > 0$ の場合であって、$f(0) = 1$ と決められていたから `else` の場合は `1` を返すのである。

数学において、再帰的に定義できるもっとも分かりやすいであろう例は、数列である。数列では**漸化式（ぜんかしき）**と呼ばれる記述の仕方がある。たとえば

$$a_{n+1} = a_n + a_{n-1}, \quad a_0 = 1,\ a_1 = 1$$

と書けば、式の記述からある**項**の値は、ひとつ前の項と 2 つ前の項の和であることが分かる。最初の 2 項が 1, 1 であることから、この例は

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

と続く。実はこの数列は**フィボナッチ**[1]**数列**として知られた数列なのである。

$f(0) = 1$ から始まる、いまの定義を **Haskell** のスクリプトにしてみよう。

(ghci env.)
```
Prelude> let f n = if n > 1 then f (n-1) + f (n-2) else 1
Prelude> f 4
5
Prelude> f 10
89
```

実際の漸化式は $f(n+1) = f(n) + f(n-1)$、$f(0) = 1$、$f(1) = 1$ であっても、スクリプトを書くならば n をひとつずらして、$f(n) = f(n-1) + f(n-2)$ と考えよう。したがって関数の定義は

[1] フィボナッチ（1174?–1250?）：イタリアの数学者。本名、ピサのレオナルドの通称。

`f n = f (n-1) + f (n-2)` となる。しかし、いずれにしても `f 0 = 1` が**初項**であることに注意されたい。このことは、$f(n+1) = \cdots$ の式を用いる場合は $n = 1$ から代入を始めて $f(2) = f(1)+f(0)$ を求めるが、$f(n) = \cdots$ の式を用いる場合は $n = 2$ から代入を始めて $f(2) = f(1) + f(0)$ を求めている。結局のところ、`f 10` という指定は第 11 項を表しているので、89 が出力されているのである。

ところで関数名の定義が `f` だけでは味気ないと思うだろうが、対話モードはキーボードからの直接入力なので、分かりやすい名付けより簡単に入力できることを優先した。このあとからは、もう少し気の利いた名前にしよう。

それよりも、再帰的な定義がされた関数がどのように動作しているかを説明するために、最初の `f 4` の処理を追ってみよう。

`n = 4` だから `if n > 1` の条件より `f (n-1) + f (n-2)` の処理が選択され、`n = 4` より具体的に `f 3 + f 2` が計算されるはずである。しかし `f 3` というのは、関数 `f n` に `n = 3` が与えられたものであり、`n > 1` の条件を満たすから、`f 2 + f 1` が計算されるはずである。おいおい、これじゃ終わらないんじゃない？

でも心配無用である。`f 2` もたしかに `f 1 + f 0` にされてしまうが、`f 1` と `f 0` は条件 `n > 1` を満たさないので、1 が返ってくる。すなわち `f 2` は `1 + 1` を返すのである。結局のところ `{f 2} + f 1` は `{1 + 1} + 1` を返すのである。つまり 3 だ。

`f 4` の処理をまとめておこう。何が何に分割されたか分かるように “`{〜}`” で区切っていることに注意されたい。

```
f 4  =  {f 3} + {f 2}
     =  {f 2 + f 1} + {f 1 + f 0}
     =  {(f 1 + f 0) + 1} + {1 + 1}
     =  {(1 + 1) + 1} + {1 + 1}
     =  5
```

見て分かるように、`f (n-1)` と `f (n-2)` は基本的に倍々に分割処理される。当然処理には時間がかかる。だから軽い気持ちで `f 40` 程度の計算をさせようとしてはいけない。理由はこうだ。

`f 1` や `f 0` にたどり着けば、この先の処理はしないので、`f 40` が倍々に処理する層が 2^{40} になるわけではない。実際はもっと少なく、2^{27} 回ほどである。一口に 2^{27} 回と言うものの、これでも軽く 1 億回を超えている。1 秒間に 1 億回の処理が可能なコンピュータでも 1 秒はかかるので、“

即座” に結果が出るわけではない。われわれが普段使っているコンピュータなら、少なくとも数十秒は要するはずだ。再帰の特性は心に留めておこう。

ところで、今回はフィボナッチ数列を

$$a_0,\ a_1,\ \ldots,\ a_9 = 55,\ a_{10} = 89,\ \ldots$$

と見たわけだが、a_0 から始まるため a_{10} は第 11 項となる。もし、a_{10} なのに11 番目というのがキモチ悪ければ

$$a_1,\ a_2,\ \ldots,\ a_{10} = 55,\ a_{11} = 89,\ \ldots$$

と見ればよい。これなら a_{10} は10 番目のフィボナッチ数を表す。したがってスクリプトは、`if n > 1` のところを `if n > 2` とすることになる。

プログラム的には “0 始まり” が自然だが、日常感覚的には “1 始まり” が普通かもしれない。庭いじりは好みのスタイルでよいだろう。