再帰と円周率

さて、中 1 から中 3 までの数学をざっと見渡したところだが、この程度の知識にちょいとトッピングを施して数学とプログラミングを楽しむことにしよう。この先は、主に**再帰**の考えと SwiftUI の組み合わせで、授業では扱わないけど中学数学の知識で理解できるような話をしてみたい。再帰は、数学的な見方・考え方を体現するのにちょうどよい教材と思われる。たとえば、数列における**漸化式（ぜんかしき）**はそれにあたる。これについては、少し先で述べることにする。

中学校程度の数学の例なら、確率の計算や場合の数を求めるとき、$5\times4\times3\times2\times1$ のような計算をすることがある。これは、たとえば『5 枚のカード$\boxed{1}$, $\boxed{2}$, $\boxed{3}$, $\boxed{4}$, $\boxed{5}$を全部使って、できる 5 桁の整数は何通りか』と問われた際にする計算である。一の位から順にカードをあてがうとすると、一の位は 5 通りから選ぶことができ、十の位は 4 通りから選ぶことができ、...、万の位は残った 1 通りを選ぶことになるので、**積の法則**より $5\times\cdots\times1=120$（通り）となる。このとき、$5\times\cdots\times1$ を 5!（5 の**階乗（かいじょう）**という）で表すことが多い。一般に

$$n!=n\times(n-1)\times(n-2)\times\cdots\times1$$

である。$1\times\cdots\times n$ の順に書くこともあるが同じことである。

ところで、たとえば 10! を求めようとすれば $10\times\cdots\times1$ を計算するのだが、もし事前に 9! の計算が済んでいたら改めて 10 から 1 まで掛けることはせず、単に $10\times$(9!の結果) をすればよい。これが再帰の基本的な考えである。要するに、『ある状態（X）を知るには、ある状態の一つ前の状態（A）を用いて、X と A との関連から X を知ることができる』ということである。

この考えをプログラミングに用いることで、プログラムは簡略に書ける場合がある。SwiftUI のサンプルコードを変更しながら見ていこう。

まず、階乗計算を再帰を用いて行う場合の仕組みは、$n!$ の計算のために $(n-1)!$ を利用するのであった。つまり『$n!$ を求める関数は、$(n-1)!$ を求める計算式を含んでいればよい』ことになる。しかし $n=1$ のとき、つまり 1! には一つ前の計算はないので、計算せずに 1 がわかる。

さて、$n!$ を求める関数を $f(n)$ と表そう。つまり $f(n)=n\times\cdots\times1$ である。すると $f(n-1)=(n-1)\times\cdots\times1$ と書ける。この関係を図式表現すると

$$n!=f(n)=\overbrace{n\times\underbrace{(n-1)\times\cdots\times1}_{f(n-1)\text{ を求める式}}}^{f(n)\text{ を求める式}}\qquad\text{および}\qquad f(1)=1$$

であるから、$f(n)=n\times f(n-1)$ になっていることがわかるだろう。ただし $n=1$ のときは、$f(1)$ と $f(0)$ の関係にならず単に $f(1)=1$ である。

これを Swift の関数（関数名は `factorial` にした）で表現すると

```
func factorial(n: Int) -> Int {
    if n > 1 {
        return n * factorial(n: n-1)
    } else {
        return 1
    }
}
```

となる。関数の中に同じ名前の関数が入っているのだが、こういう書式であると思ってもらいたい。

この関数を、あらかじめ表示されたサンプルコードに追加して、さらに “Hello, world!” を出力する部分を書き換えれば出来上がりだ。変数 `n:` に与える数を変えれば、その数の階乗が App Preview に表示される[1]。

[factorial.playground]

```
 1 import SwiftUI
 2
 3 struct ContentView: View {
 4     var body: some View {
 5         VStack {
 6             Text("\(factorial(n: 10))")
 7         }
 8     }
 9
10     func factorial(n: Int) -> Int {
11         if n > 1 {
12             return n * factorial(n: n-1)
13         } else {
14             return 1
15         }
16     }
17 }
```

コード中の `var body:` や `VStack {` など、気になる書式はあるだろうが、詳しい勉強は各自でお願いしたい。ここでは、サンプルコードを「雛型（ひながた）」として使っているだけなのだから。この先は、目的に沿うようにコードを追加・修正していくことにする。

ところで、いまは単に $n!$ の計算だけをしたのだが、次のような計算をするとどうなるだろうか。

$$1+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\frac{1}{4!}+\frac{1}{5!}+\cdots+\frac{1}{n!}+\cdots$$

[1] App Preview が表示されていなければ、右上の アイコンのプルダウンメニューから “Canvas” にチェックを入れよう（するとアイコンは に変わる）。

先頭の “$1+$” が他の項と少し異なり違和感を持つだろうが、そのことは後で説明しよう。それと、右端が “$+\cdots$” となっていることから、この式は終わりがない。終わらない足し算で何らかの答が得られるか気になるけれど、この場合はきちんと値が確定する。その値は $2.71828\cdots$ であり、円周率同様、小数点以下無限に続く数である。**自然対数の底（てい）**と呼ばれ、円周率以上に重要な意味を持つ定数である。指数関数・対数関数に絡（から）んで登場する。円周率が π で表されるように、自然対数の底は e で表す。

もっともコンピュータで無限に計算をするわけにはいかないので、ある程度の項で計算を打ち切る必要はある。では、実際に計算をしてみよう。

[evalue.playground]

```
 1 import SwiftUI
 2
 3 struct ContentView: View {
 4     var body: some View {
 5         VStack {
 6             Text("\(evalue(n: 17), specifier: "%.15f")")
 7         }
 8     }
 9
10     func evalue(n: Int) -> Double {
11         if n > 1 {
12             return evalue(n: n-1) + 1 / Double(factorial(n: n))
13         } else {
14             return 1
15         }
16     }
17
18     func factorial(n: Int) -> Int {
19         if n > 1 {
20             return n * factorial(n: n-1)
21         } else {
22             return 1
23         }
24     }
25 }
```

ここでは、先のプログラムに再帰関数 `evalue()` を追加して、出力の `Text()` の書式を少し変えただけである。ただし、再帰関数の仕組み

$$f(n) = \overbrace{\underbrace{\frac{1}{1!}+\frac{1}{2!}+\cdots+\frac{1}{(n-1)!}}_{f(n-1)\text{ を求める式}}+\frac{1}{n!}}^{f(n)\text{ を求める式}} \qquad \text{および} \qquad f(1)=1$$

を明確にするために先頭の “1 +” は計算に含めていない。したがって、ここで計算される値は $1.71828\cdots$ となる。この場合、14 行目の `return 1` は $f(1) = \dfrac{1}{1!}$ の値である。

もし 11 行目を `if n > 0` とすれば、計算結果は $2.71828\cdots$ となる。なぜなら、`n = 1` のとき `evalue(n: 0) + 1 / Double(factorial(n: 1))` によって返される `evalue(0) := 1` は $f(0)$ を意味する。$f(0)$ は $f(n)$ の定義にはないが、$f(0)$ が $f(n)$ に含めなかった先頭の “1 +” を意味すると考えれば納得できるだろう。

仕組みの図式表現から $f(n) = f(n-1) + \dfrac{1}{n!}$ がわかるので、関数 `evalue()` が上記のように書けることはよいだろう。ただし分数計算をしているので、適宜 `Double` を加える必要がある。

ところで $f(n) = f(n-1) + \dfrac{1}{n!}$ のように、$f(n)$ と $f(n-1)$ の関係を式で表したものが先に述べた漸化式である。この場合はさらに**初期値**として $f(1) = 1$ も与えられている。数列を学習すると、たとえば漸化式を『$a_n = 2a_{n-1}$、**初項** $a_1 = 1$』のように書くことが多い。a_n は $f(n)$ に対応した書き方である。どちらも同じ意味を持つが、プログラミングの学習では $f(n)$ のように書く方が扱いやすいと思う。

漸化式と再帰関数の例として、**フィボナッチ数列**[2]を求めるプログラムも書いておこう。フィボナッチ数列は『直前の 2 項の和を次の項とする』規則で作られる。したがって、図式表現では

$$\underbrace{1,\ 1,}_{\text{最初の 2 項}}\ 2,\ 3,\ 5,\ 8,\ \ldots,\ \underbrace{(\overbrace{(n-2)\text{ 項目の値}}^{f(n-2)\text{ を求める式}}),}\ \underbrace{(\overbrace{(n-1)\text{ 項目の値}}^{f(n-1)\text{ を求める式}}),}\ \overbrace{(n\text{ 項目の値})}^{f(n)\text{ を求める式}},\ \ldots$$

である。漸化式なら $f(n) = f(n-1) + f(n-2)$、$f(1) = 1$、$f(2) = 1$ と書くところである。この関係を再帰関数として書き出せば

```
func fibonacci(n: Int) -> Int {
    if n > 2 {
        return fibonacci(n: n-1) + fibonacci(n: n-2)
    } else {
        return 1
    }
}
```

となる。`if` 文が `n > 2` であるのは、それが 3 項目以降の関係式であり、1, 2 項目が該当しないからである。これで、たとえば

```
Text("\(fibonacci(n: 30))")
```

[2] レオナルド=フィボナッチ (1170?–1250?) : イタリアの数学者。本名はレオナルド・ダ・ピサ（ピサのレオナルド）と言われている。

とすれば、30 項目のフィボナッチ数が計算される。このとき、もしかしたら計算結果が表示されるまでの時間がちょっと長いと感じるかもしれない。実際、長いのである。

というのは、再帰関数は記述は簡便であるものの、たとえば `fibonacci(n: 30)` を行う場合、まず `fibonacci(n: 29) + fobonacci(n: 28)` に展開される。すると今度は `fibonacci(n: 29)` が `fibonacci(n: 28) + fibonacci(n: 27)` に展開され...、すべての項が `fibonacci(n: 2)` または `fibonacci(n: 1)` になるまで繰り返される。この総数は数億回にもなるので、当然計算時間はかかることになる。プログラムに再帰関数を使う際は気をつけないといけない。

最後に、円周率の計算を再帰関数を用いてやっておこう。円周率を求めるための式は、さまざまな形で与えられている。もっとも簡単な関係式は

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \cdots (\pm)\frac{1}{\text{奇数}}(\mp)\cdots$$

であるが、加減する項が “偶数番目の項まで” か “奇数番目の項まで” かで扱いが異なる。図式表現をすると

$$n \text{ が偶数：} \quad \frac{\pi}{4} = \overbrace{\underbrace{1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + \frac{1}{2(n-1)-1}}_{f(n-1)\text{ を求める式}} - \frac{1}{2n-1}}^{f(n)\text{ を求める式}} + \cdots$$

$$n \text{ が奇数：} \quad \frac{\pi}{4} = \overbrace{\underbrace{1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots - \frac{1}{2(n-1)-1}}_{f(n-1)\text{ を求める式}} + \frac{1}{2n-1}}^{f(n)\text{ を求める式}} - \cdots$$

であるから、漸化式は $f(n) = f(n-1) - \dfrac{1}{2n-1}$ か $f(n) = f(n-1) + \dfrac{1}{2n-1}$ の 2 通りを考えなくてはならない。

そういうことなら再帰関数は、たとえば

```
func pivalue(n: Int) -> Double {
    var sgn: Double
    if n % 2 == 0 { sgn = -1 } else { sgn = 1 }

    if n > 1 {
        return pivalue(n: n-1) + sgn / Double(2*n-1)
    } else {
        return 1
    }
}
```

とするのはどうだろうか。実際この関数を用いて、たとえば

```
Text("\(4 * pivalue(n: 10000), specifier: "%.15f")")
```

とすれば、10000 項までの和で円周率が表示される。と言っても、精確な値 3.141592653589793 にはほど遠い。それも当然で、この計算式では 10000 項近辺の精度は 0.0001 でしかない。16 桁分の精度など夢のような話なのである。

そこで普通は、もっと速く**収束**する式を用いるのだが、そういう式は Internet からいくらでも探せるので、各自で調べるとよいだろう。ただし、ここに提示したプログラムの式を差し替えるだけでは、結局 `Double` 型の精度でしか計算できないので、たとえば 100 桁の精度で円周率を求めるわけにはいかない。それは別の方法を用いることになるが、この Site 内でもいくらかの方法を試している。いまは、SwiftUI で簡単な再帰関数を用いてコードを書くことが目的なので、高精度で円周率を計算するプログラムは別のところをあたってもらいたい[3]。

[3]ちなみに**モンテカルロ法**で円周率の近似値を求めるのは、プログラミングの練習にはなっても、円周率の値としては意味がない。