

## 第 12 章 ストリームによる計算 (Computing with Streams)

前の章で、変数、代入、および状態を持つオブジェクトを紹介しました。時間とともに変化する実世界のオブジェクトを、計算において変数の状態を変化させることで、モデル化できることを見ました。実世界では時刻は変化します。そのように、プログラム実行における時刻の変化によって、実世界の時刻変化をモデル化できます。もちろん、そのような時刻変化は、通常伸びたり縮んだりしますが、相対的な順番は守られます。これはまったく自然に見えますが、注意すべき大事なことがあります。いったん変数と代入を導入すると、我々のシンプルでパワフルな関数ベースの計算の置き換えモデルは、もはや適用できないのです。

ほかに方法はないのでしょうか？ 実世界の状態変化を、状態を持たない関数を使ってモデル化できないのでしょうか？ 数学の導きによると、答えは明らかに Yes です。時間変化する量は、時刻  $t$  をパラメータにとる関数  $f(t)$  によって、シンプルにモデル化できます。モデル化だけでなく計算においても、これはうまくいきます。変数を次々と書き換える代わりに、それらすべての値をリストの連続的な要素として表現できます。つまり、ミュータブルな変数  $\text{var } x: T$  は、イミュータブルの値  $\text{val } x: \text{List}[T]$  で置き換えることができます。ある意味、空間と時間の取引です。変数に代入される様々な値は、リスト中に異なる要素として同時に存在することになります。リストベースモデルの利点の一つは、「タイムトラベル」、つまり変数に代入される連続的な値を同時に見ることで、できることです。ほかの利点としては、強力なリスト処理関数ライブラリを利用して、しばしば計算をシンプルにできることです。たとえば、特定の範囲にある素数すべての和を計算する、命令型プログラムを考えてみましょう。

```
def sumPrimes(start: Int, end: Int): Int = {
  var i = start
  var acc = 0
  while (i < end) {
    if (isPrime(i)) acc += i
    i += 1
  }
  acc
}
```

変数  $i$  が、範囲  $[\text{start} .. \text{end}-1]$  のすべての値を「経験」していることに注意してください。より関数的な方法では、変数  $i$  の値のリストを  $\text{range}(\text{start}, \text{end})$  によって直接に表現します。

```
def sumPrimes(start: Int, end: Int) =
  sum(range(start, end) filter isPrime)
```

あきらかにプログラムは短くて明快になりました！しかし、この関数型プログラムは効率の点でかなり劣ります。範囲内のすべての数からなるリストを作り、さらにそのうちの素数すべてからなるリストを作るからです。効率の点ではさらに悪いことがあります。次の例を見てください。

1000から10000の間の二番目の素数を見つける。

```
range(1000, 10000) filter isPrime at 1
```

これは、1000 から 10000 までのすべての数からなるリストが作りますが、そのリストのほとんどの要素は顧みられません！しかし、あるトリックによって、これらのような例を効率的に実行できます。

シーケンスの後の要素が実際には計算に必要なければ、その計算を回避できるのです。

このようなシーケンスのために新しいクラスを定義します。それを `Stream` と呼びます。Stream は定数 `empty` とコンストラクタ `cons` を使って生成します。これらは `scala.Stream` モジュールで定義されています。たとえば、次の式は 1 と 2 を要素とするストリームを生成します。

```
Stream.cons(1, Stream.cons(2, Stream.empty))
```

ほかの例として、`List.range` の類似物、ただしリストの代わりにストリームを返すものは、次のようになります。

```
def range(start: Int, end: Int): Stream[Int] =
  if (start >= end) Stream.empty
  else Stream.cons(start, range(start + 1, end))
```

(この関数は、上のモジュール `Stream` でも定義されています) `Stream.range` と `List.range` は似ていますが、その実行時の振る舞いはまったく違います。

Stream.range は、最初の要素が start である Stream オブジェクトを直ちに返します。そのほかのすべての要素は、tail メソッド呼び出しによって、それらが **必要** になったときにのみ計算されます (tail メソッドはまったく呼ばれないかもしれません)。

ストリームは単なるリストとしてアクセスされます。リストと同様、基本的なアクセスメソッドは isEmpty と head と tail です。たとえば次のようにして、ストリームのすべての要素を表示できます。

```
def print(xs: Stream[A]) {  
  if (!xs.isEmpty) { Console.println(xs.head); print(xs.tail) }  
}
```

ストリームは、リストに対して定義されている他のほぼすべてのメソッドもサポートしています (これらがサポートするメソッドの差分については、下記を参照してください)。たとえば、1000 から 10000 の範囲のストリームに filter と apply を適用して、1000 から 10000 の間の二番目の素数を見つけることができます。

```
Stream.range(1000, 10000) filter isPrime at 1
```

先のリストベースの実装との違いは、もはや不必要に三番目以降を構築して素数判定しないことです。

**ストリームの CONS と連結** クラス List のメソッドのうち、クラス Stream ではサポートされていないものは、:: と ::: の 2 つです。理由は、これらのメソッドは右側の引数に対して呼ばれるからです。右側の引数に対して呼ばれるということは、その引数は、メソッドが呼ばれる前に評価される必要があるということです。たとえばリストの x :: xs の場合、後部 xs は :: が呼ばれる前に評価される必要があり、新しいリストが構築される場合があります。これはストリームではうまくいきません。ストリームの後部は、それが tail オペレーションによって必要となるまでは評価されてはなりません。リストの連結 ::: をストリームに持ち込めないのも、同じ理由です。

x :: xs の代わりに、最初の要素 x と (未評価の) 後部 xs からなるストリームを構築するには、Stream.cons(x, xs) を使います。xs ::: ys の代わりに、オペレーション xs append ys を使います。

[前ページ](#) [12章](#) [目次](#) [次ページ](#)

名前:	<input type="text"/>
コメント:	<input type="text"/>

投稿