2.2 関数

さて、ふらふらと翔んでいる最中だけれど、Lua の関数の話をもう少し続けよう。いますぐに必要ではないけれど、あとで使うことになる Lua の便利な関数機能を説明したい。フィボナッチ数列の隣り合う 2 項の比を計算したとき、$\frac{(後の項)}{(前の項)}$ で計算したよね。でも、比なんだから $\frac{(前の項)}{(後の項)}$ で計算してもよいはずだ。この違いは、実は面白い景色なので見ることにしよう。

早速こんな関数『

```
\begin{luacode*}
function fibNs(n)
    a = 1  b = 1
    for i = 2, n do
        a, b = b, a + b
    end
    return a, b
end
\end{luacode*}
```

』を定義しておこう。`return` 文を見て分かるように、同時に 2 個の値を返している。そのためこの関数を『`\directlua{n0, n1 = fibNs(10) tex.print(n0 .. ', ' .. n1)}`』のように使うと『55, 89』のように 2 個の値が出力される。スクリプトから察せられるように、フィボナッチ数列の第 n 項と第 $n+1$ 項を一緒に出力する関数である。

これを使ってり隣り合う 2 項の比を $\left(\frac{(後の項)}{(前の項)}, \frac{(前の項)}{(後の項)}\right)$ のように組にして出力してみよう。スクリプトは『

```
\[\begin{array}{llll}
\directlua{
    for i = 1, 8 do
        for j = 1, 2 do
            prev, post = fibNs(2*(i-1) + j)
            tex.print('(' .. post / prev .. ', ' .. prev / post .. ')'
                                                    .. \asluastring{&})
        end
        tex.print(\asluastring{\\[-5pt]})
    end
    tex.print(\asluastring{\dots \\})
}
\end{array}\]
```

』のように書いて、その結果は『

$(1.0, 1.0)$ $(2.0, 0.5)$
$(1.5, 0.66666666666667)$ $(1.6666666666667, 0.6)$
$(1.6, 0.625)$ $(1.625, 0.61538461538462)$
$(1.6153846153846, 0.61904761904762)$ $(1.6190476190476, 0.61764705882353)$
$(1.6176470588235, 0.61818181818182)$ $(1.6181818181818, 0.61797752808989)$
$(1.6179775280899, 0.61805555555556)$ $(1.6180555555556, 0.61802575107296)$
$(1.618025751073, 0.61803713527851)$ $(1.6180371352785, 0.61803278688525)$
$(1.6180327868852, 0.61803444782168)$ $(1.6180344478217, 0.61803381340013)$
...

』である。何か気づかないだろうか。

$\left(\frac{\text{(後の項)}}{\text{(前の項)}}, \frac{\text{(前の項)}}{\text{(後の項)}}\right)$ の 2 数には 1 程度の違いがある。もっと先まで見ればはっきりするが、紙面を無駄に使うこともない。結論を言えば、先へ行くほど差は 1 に近づく。どういうこと？　それはもうしばらく先の路で説明しよう。

ところで関数と Lua の振る舞いについては知っておくべきことがある。簡単な例で説明しよう。まず、

```
\begin{luacode*}
    n = 10      n = n * 3      tex.print(n)
\end{luacode*}
```

を処理すれば 30 が出力される。その直後に

```
\begin{luacode*}
    n = n * 5      tex.print(n)
\end{luacode*}
```

を処理すれば 150 が出力される。あとのコードでは `n` に値が代入されていないが、その直前の処理で `n = 30` になっていたのでそれが使われたのだ。`n` は `luacode*`環境の中だけで有効なのではなく、TEX からは二つの `luacode*`環境は地続きのように見えている。もし、一番目の `luacode*`環境が後ろに影響しないようにしたければ、TEX では一般的な `{, }` による**グルーピング**をすればよい。つまり `luacode*`環境全体を `{, }` で囲むことである。

しかしそうすると、二番目の `luacode*`環境で `n` が初期化されてないためエラーとなる。Lua のこの振る舞いは大変よい。なぜなら、変数は初期化して使うことを強要されるので、知らぬ間に変な値が代入されていることが防げるからだ。ちなみに一番目の `luacode*`環境において

```
n = 10   ⇒   local n = 10
```

のようにして代入すると、TEX のグルーピングがなくても n は一番目の luacode*環境内だけで有効な**ローカル変数**となる。したがって二番目の luacode*環境で n が初期化されずエラーとなる。

Lua の変数は local を明示しなければ**グローバル変数**で扱われる。これには注意がいる。

```
x  = 1  tex.print(x .. ' ')              ...(A)
for i = 1, 1 do
    x = 2  tex.print(x .. ' ')           ...(B)
    for j = 1, 1 do
        local x = 3  tex.print(x .. ' ') ...(C)
    end
    tex.print(x .. ' ')                  ...(D)
    x = 3  tex.print(x .. ' ')           ...(E)
end
tex.print(x)                             ...(F)
```

（結果）1 2 3 2 3 3

上のテストプログラムと結果を見てほしい。for 文は繰り返しではなく、for〜end 内部で x が再定義されたことを示すために用いている。A, B, C, E の出力は妥当だろう。x への代入直後の値を示しているからだ。

D の出力が 2 であるのは、その前の x が local で定義されたからである。for などのループ内において local で定義された変数は、end までのループ内が有効範囲となる。したがって、D で出力される x の値は for j 内で local 定義された x ではなく、その前 B で定義された値が出力されているのだ。

一方 F の出力が 3 であるのは、E で定義された x が local でない、すなわちグローバルな変数であったため、直前の代入値が出力されている。ちなみに、E の代入を local x = 3 とすると、x は for i 内で有効な変数となるが、それ以前に for i 内で x = 2 がグローバル変数であったので 2 が出力される。

説明がぎこちなくて申し訳ない。Lua の変数の振る舞いについては十分調べて理解してもらいたい。ただ、この振る舞いは TEX には都合よいものと思われる。

ところで、関数の作り方に規則はないのだろうか。実は規則や注意すべき点は山ほどある。しかし、私はこの夢見鳥の路を翔び回りながら、生産的なソフトウェアを作ったり効率的なプログラムを書いているわけではない。だから関数の作り方もいい加減なもので、幸運にも動いていると言う

方が的を射ている。Lua において関数の作り方のガイドラインは存在しているに違いない。そして私には無関係なだけなのだ。

しかし関数の作り方の基本は、一つの関数に多くの機能を持たせないことだ。一つの関数には一つの機能が望ましい。

一つの関数に多くの機能を持たせると、トラブルが発生したとき、関数内のどこが問題であるかを特定しづらい。もし、二つの関数に分けて書けるなら是非そうしよう。なぜなら、トラブルの原因を特定しやすいからだ。関数には勢いよく機能を詰めがちだが、自分のためにもいくつかの小さな関数に分けるほうがよい。

そうは言っても、私はそこまで真剣に考えながら翔び回っていない。もともと書くスクリプトが小さいことと、これをもとに何かを作るわけではないからだ。あくまでも気まぐれにふらふらしていると思ってもらいたい。