2.3 再帰関数

Haskell の対話モードは手軽で便利なのだが、規模が大きいスクリプトを書くのには向いていない。1 行ずつ `let` で定義するのは、うっかりタイプミスをした場合などに面倒極まりない。ここからは必要に応じてファイルにスクリプトを保存しながら実行していこう。修正は楽だし、ひとつのファイルに関数を書きためておくこともできる。

次のスクリプトをたとえば `garden.hs` という名前で保存してみよう。庭いじりをしているから `garden` と名付けたが、実際は何でもよい。しかし拡張子は **Haskell** のファイルであるから `.hs` にしておこう。分かっていると思うけど、スクリプトはプロンプト `Prelude>` が出ているところで書くのではなくて、適当なテキストエディタを起動して編集するんだよ。

[hs script]
```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n = n * factorial (n-1)
```

さて、保存したファイルを実行するわけだけれど、そのためには GHC をあらためて起動しなくてはならない。重要なことは、保存したファイルがあるディレクトリで GHC を起動することである。たとえばファイルが `c:\myfolder\myscript` ディレクトリにあれば、`cd c:\myfolder\myscript` のように打ち込んで、`myscript` ディレクトリに移動しておかなくてはいけない。その上で GHC を起動して、画面に `Prelude>` が表示されたら `:l garden` と入力する。`:l` はファイルを読み込むための命令（コマンド）だ。すると

(ghci env.)
```
Prelude> :l garden
[1 of 1] Compiling Main             ( garden.hs, interpreted )
Ok, one module loaded.
*Main>
```

のように、ファイルに保存した関数がロードされる。

もし、このときファイルがロードされないで **Haskell** が何やら文句を言ってきたら、ロードするファイル名を間違えたか、またはスクリプトが正しく書けていないのだ。ファイル名の間違えなら再度正しいファイル名で読み込ませればよいが、スクリプトに間違いがあったら、面倒でもファイルを編集し直して保存する。そして、もう一度ファイルをロードし直すのだ。

ファイルが読み込まれると、プロンプトは`*Main>` になるが、環境により違うことがあるかもしれない。今後は、対話モードで実行する場合とファイルをロードして実行する場合があるので、どっちで実行しているかはプロンプトの表示で判断してもらいたい。でも、`*Main>` が表示されて

いれば対話モードも OK なんだけどね。さあ、ここで `factorial 10` を実行してみよう。

(ghci env.)

```
*Main> factorial 10
3628800
```

ちゃんと 10! の値が表示されたね。たった 3 行のスクリプトだが、簡単に説明しておこう。

1 行目の `factorial :: Integer -> Integer` は「関数 `factorial` は、整数型（Integer 型）の値を受け取り、整数型（Integer 型）の値を返す」と読めばよい。この行がなくても関数は正常に動作するが、**Haskell** では型を明示することはよい習慣とされている。

2 行目の `factorial 0 = 1` は、再帰関数の基点となる定義である。つまり `factorial 0` は `1` を返す、と定義したのだ。“=” の使い方は最初違和感があるだろうが、関数や式の定義のために使う記号である。`0` と `1` が等しいと読まないように。**Haskell** において等号は “==” だったよね。

3 行目が関数 `factorial n` の実質的定義だ。`factorial n` が $f(n) = n \times f(n-1)$ の式で再帰的に定義されたことが分かるはずだ。

もう少し再帰について話してみたい。階乗（$n!$）の計算が出たところで、唐突だが

$$\frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \cdots\cdots + \frac{1}{n!}$$

の計算を考えてみよう。実のところこの式は $n \to \infty$ のとき一定の値に**収束**する。それは**自然対数の底**といい、円周率 π に匹敵、いやそれ以上に重要な値である。有効桁数 16 桁で表せば 2.718281828459045 だ。真の値は π と同様に**閉じた式**で表せないので、近似値もしくは e で表している。

さて、e の近似値を計算するには再帰の考えが使える。まず

$$f(n) = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \frac{1}{4!} + \frac{1}{5!} + \cdots\cdots + \frac{1}{(n-1)!} + \frac{1}{n!}$$

とおいてみる。式の形から $\dfrac{1}{n!}$ の 1 項手前までの和は $f(n-1)$ と同じことである。つまり、$f(n) = f(n-1) + \dfrac{1}{n!}$ だから、$f(n)$ が $f(n-1)$ を用いて表せた。$f(0) = 1$ であることから、**Haskell** で $f(n)$ を定義すると次のようになる。

[hs script]

```
evalue :: Integer -> Integer
evalue 0 = 10^1000
evalue n = evalue (n-1) + 10^1000 `div` (factorial n)
```

このスクリプトは、さっき保存したファイル `garden.hs` に記述した `factorial` 関数の下に書き加えてほしい。その上でファイルをロードし直して `evalue 449` を実行しよう。

(ghci env.)

```
*Main> :l garden
[1 of 1] Compiling Main             ( garden.hs, interpreted )
Ok, one module loaded.
*Main> evalue 449
27182818284590452353602874713526624977572470936999595749669676277240766303
53547594571382178525166427427466391932003059921817413596629043572900334295
26059563073813232862794349076323382988075319525101901157383418793070215408
91499348841675092447614606680822648001684774118537423454424371075390777449
92069551702761838606261331384583000752044933826560297606737113200709328709
12744374704723069697720931014169283681902551510865746377211125238978442505
69536967707854499696979468644549059879316368892300987931277361782154249992
29576351482208269895193668033182528869398496465105820939239829488793320362
50944311730123819706841614039701983767932068328237646480429531180232878250
98194558153017567173613320698112509961818815930416903515988885193458072738
66738589422879228499892086805825749279610484198444363463244968487560233624
82704197862320900216099023530436994184914631409343173814364054625315209618
36908887070167683964243781405927145635490613031072085103837505101157477041
71898610687396965521267154688957035011 6
```

ほとんど瞬時に e の値 1000 桁が表示される。もっとも、先頭の 27 のところは `2.7` と読み替えてもらうんだけど。

なぜ小数点が表示されないかというと、このスクリプトは e の小数点以下 1000 桁までを計算したのではなく、$10^{1000} \times e$ の値を計算したからだ。理由は **Haskell** が、小数点以下の有効数字は 16 桁程度しか保証しないが、整数値の上限はないに等しいからだ。もちろん正確な 1000 桁分の e の値を、このあと計算に使いたいというなら大いに困る。しかし、単に 2.7 の先の数字がどうなっているかに関心があるだけなら、小数点の有無なんて関係ないだろう。だからスクリプトは、最初に `evalue 0 = 10^1000` と定義したのである。10^{1000} は本当は 1001 桁だけどね。

肝心の計算は 1 行しかない。`evalue n = evalue (n-1) + 10^1000 ‘div‘ (factorial n)` を見れば $f(n) = f(n-1) + \dfrac{10^{1000}}{n!}$ を記述していることが分かるだろう。割り算のために‘`div`‘を用いていることに注意してほしい。演算子 “`/`” を使ってしまうと小数値で計算されてしまう。というより、`Integer -> Integer` の型に合わずエラーとなる。ちなみに “ ‘ ” はバッククォートであり、アポストロフィ “ ’ ” ではない。間違えないように。

ところで `evalue 449` の引数 449 には何か意味があるのだろうか。そう、多いにワケアリなのだ。それは $n = 449$ のとき、$n!$ が初めて 1000 桁を超えるからだ。1000 桁の精度で計算をしたかったのだから、$\dfrac{1}{n!}$ が $\dfrac{1}{10^{1000}}$ を下回ったら計算する意味がないからね。

449 を特定する計算の仕方はこうである。要するに $n! > 10^{1000}$ となる n が分かればよいので、

4

不等式を解くために両辺とも 10 の対数をとって

$$\log_{10} n! > \log_{10} 10^{1000}$$

を考える。$n! = n(n-1)(n-2)\cdots 1$ と対数の性質 $\log_{10} MN = \log_{10} M + \log_{10} N$ から、不等式は

$$\log_{10} n + \log_{10}(n-1) + \log_{10}(n-2) + \cdots + \log_{10} 1 > 1000$$

となる。さすがにこれを簡単に解くわけにはいかないが、対数表と電卓が用意できれば何とか計算できる。でも、表計算ソフトで計算する方が楽だろう。実際にこれを満たす n を調べると、$n = 449$ で和が 1000.238891 になることが分かるのである。