■ 連載

Haskell プログラミング

関数プログラミングの妙味

和田 英一 (IIJ 技術研究所)
wada@u-tokyo.ac.jp

連載開始にあたって

3月までの連載，プログラム・プロムナードはICPC（ACM国際大学対抗プログラミングコンテスト）の問題を解いてみせ，ICPCへの参加を奨励しようとするものであった．したがって学会の電子図書館でもプロムナードはだれでも見られる．ところが連載が3年も続くと，すでに取り上げたのに似たような問題が多く残り，執筆陣の士気も振わなくなってきた．

一方，ACMは別のプログラミングコンテストも開催している．ICFP (International Conference on Functional Programming) のProgramming Contestといい，第1回は1998年にMITで開かれた．2002年のコンテストでは東大米澤研のメンバが優勝している．

ICPCはある会場に集まり，5時間で8題近くを解く．会場キャパシティの制約から世界大会には多くの予選を経た代表だけが集まるのだが，ICFP PCはインターネットで参加し，解答時間も72時間と大盤振舞いであり，言語はなんでもよいと鷹揚である．ただし関数プログラミングの研究会の主催だから，Haskellが増えているらしい．72時間あるということは，それなりに問題が巨大なわけで，2004年にはアリの脳をシミュレートする有限オートマトンの状態遷移図を作るというのであった[1]．

左様な次第で，4月からはICFP PCを目指し，関数プログラミング言語Haskellの話を何人かで分担して連載したい．といってもHaskellを完全に説明するのではなく，関数プログラミング言語でプログラムを書くとこのような調子になる，という感じが分かっていただければとりあえずはよい．詳しいことは必要になったら説明することになろう．なんだまた関数プログラミングかといわず，騙されたと思ってつきあってほしい．関数プログラミングはプログラミングの原点なのだから．

関数プログラムは簡潔，明快

そもそも関数プログラミングの功罪については以前から巷間にかまびすしかった．しかしその議論をここでは繰り返さない．スウェーデンChalmers工科大学のJohn Hughesの書いた "Why Functional Programming Matters" [2] があり，これを見るのが手っ取り早い．そこでの数値計算や人工知能の例題はMirandaによっており，Haskellではないが，関数プログラミングの雰囲気は十分に分かる．この文献も，山下によるその和訳もWebから取ることができる．

一方，Haskellの入門書もWebから得られる．Yale大学のPaul Hudak他による "A Gentle Introduction to Haskell 98" [3] が懇切で，これも山下の和訳とともにWebから取れる．

新しい言語を使い始めるときは，覚えることはかなりあるが，それはプログラムを巧みに書くための投資である．もちろん語学と同じで，プログラミング言語もしばらく使わないと錆びてくるから要注意．

まず面白い例から．たとえばHaskellの解説によくある例を借りればnの階乗は

```
product [1..n]
```

である．`[1..n]`は1からnまでの整数のリスト`[1,2,3,...,n]`を生成し，`product`は乗算の単位元1に引数を次々と掛ける関数である．0の階乗は`[1..0]`で，これは空リスト．したがって結果は1となる．Haskellで何種類もの階乗プログラムを書いた楽しい話はThe Evolution of a Haskell Programmer[4]にある．

Haskellのクイックソートは劇的だ．

```
quicksort  []       =  []
quicksort (x:xs)    =  quicksort [y | y <- xs, y<x]
                       ++ [x]
                       ++ quicksort [y | y <- xs, y>=x]
```

ここには定義が2つある．1行目は引数が空の場合を定義する．`[]`が空リスト．2行目は空でない引数の場合を定義する．引数`x:xs`は（空かも知れぬ）リスト`xs`の前に`xs`の要素の型の`x`を`cons`（`:`で表す）したものである．引数にこういう構造が書けるのも特徴である．右辺に現れる`++`はリストの`append`である．`[y | y <- xs, y<x]`は`xs`の要素`y`で，`y<x`なるもののリストを意味する．

したがって，空でないリスト`x:xs`のクイックソートは，先頭の要素を`x`，残りを`xs`とし，1）`xs`の要素で`x`未満のものをクイックソートしたものと，2）`x`をリストにした`[x]`と，3）`xs`の要素で`x`以上のものをクイックソートしたものを`append`すればよい．再帰的にクイックソートするリストはだんだん短くなり，最後に空リストのクイックソートは1行目の定義で空リストになる，と読む．

世の中のクイックソートに比べると，驚くほど簡単である．その理由の1つはリストの先頭の要素`x`を比較の対象にするからである．周知のように，すでにソートされているリストを再びソートすると，悲劇が起きる（リストの長さをnとしてn^2の時間がかかる）という脳天気なプログラムであることに注意しよう．しかし明快なことは理解できる．

Backusの FP

関数プログラミングの歴史には，Lispから始める文献もあるが，1977年のTuring賞を受賞したJohn BackusはCACM誌に「プログラミングはフォン・ノイマン・スタイルから解放されうるか？　関数型プログラミング・スタイルとそのプログラム代数」という長大な論文[5]を送ってFPを提案し，これを関数プログラミングの嚆矢とするのがよさそうである．von Neumannボトルネックという語もここで登場した．

しばらく記法が変わるが我慢してほしい．fをxに作用させるのをFPでは$f:x$のように書く．したがって2の階乗は$!:2$となる．その階乗の定義は

Def　$! \equiv \mathrm{eq0} \rightarrow \overline{1}; \times \circ [\mathrm{id}, ! \circ \mathrm{sub1}]$

where

Def　$\mathrm{eq0} \equiv \mathrm{eq} \circ [\mathrm{id}, \overline{0}]$

Def　$\mathrm{sub1} \equiv - \circ [\mathrm{id}, \overline{1}]$

のように書く（$\circ$は関数の合成，$\overline{0}$, $\overline{1}$などは何をもらってもそれぞれ0, 1などを返す関数）．こう定義してから2の階乗をやってみる．

$!:2$

$=$ {階乗の定義を引数2に作用させる．!の定義を代入}

$(\mathrm{eq0} \rightarrow \overline{1}; \times \circ [\mathrm{id}, ! \circ \mathrm{sub1}]):2$

$=$ {引数を分配する（$b \rightarrow c\,; a$はMcCarthyの条件式 **if** b **then** c **else** a）}

$(\mathrm{eq0} : 2) \rightarrow (\overline{1} : 2);(\times \circ [\mathrm{id}, ! \circ \mathrm{sub1}] : 2)$

ところでこの条件式の述語部分は

eq0 : 2
= {eq0 の定義を代入する }
$(\mathrm{eq} \circ [\mathrm{id}, \overline{0}]) : 2$
= {∘ は関数合成だから }
$\mathrm{eq} : ([\mathrm{id}, \overline{0}] : 2)$
= {$[f_1,\ldots,f_n] : x \equiv <f_1 : x,\ldots,f_n : x>$, 関数の並び (組立て) の引数への作用は結果が並ぶ }
$\mathrm{eq} : <\mathrm{id} : 2, \overline{0} : 2>$
={id : $x \equiv x$, $\overline{0} : x \equiv 0$ (id と $\overline{0}$ の定義)}
eq : < 2, 0 >
={eq は 2 つの引数が等しいか見る }
F … {F は偽のこと．階乗の計算の方はまだまだ続くが後略 }

Backus の論文は関数形式のプログラミングだけでなく，Laws of the Algebra of Programs というものを登場させた．任意の関数 f, g, h について

$$[f, g] \circ h \equiv [f \circ h, g \circ h]$$

つまり，関数の組立て $[f, g]$ と関数 h の合成は，関数 f と h の合成と，g と h の合成の組立てである，というように読む．もちろん証明もついている．

$([f, g] \circ h) : x$
= { 合成の定義による }
$[f, g] : (h : x)$
= { 組立ての定義による }
$<f : (h : x), g : (h : x) >$
={ 合成の定義による }
$<(f \circ h) : x, (g \circ h) : x>$
={ 組立ての定義による }
$[f \circ h, g \circ h] : x$ ▌

今ではこういう証明は当たり前だが，当時は新鮮であり驚きであった．こういうことができるのも関数プログラミングなればこそである．

その後 Robin Milner の ML(meta language の意)[6]，David Turner の Miranda[7] など，関数プログラミング言語がいくつも登場した．ML は型推論を特徴としていた．一方 Miranda は遅延評価 (lazy evaluation) と非正格 (nonstrict) 関数を提供した．

Haskell については言語設計委員会が 1987 年にでき，4 版まで更新した．1997 年におおむね完成し，似たようなものがたくさんあると，どれを使ったらよいか分からず，関数プログラミングの発展にも支障になるので，Haskell の更新作業も終結させた．そして 1998 年に規格としてまとまったのが Haskell 98 である．これからの関数プログラミング言語は Haskell 98 だけを知っていればよいことになる．Haskell 98 の報告書は Journal of Functional Programming の特別号として 2003 年 1 月に刊行され，単行本としては Cambridge University Press から出ている．Web でも見られる [8]．

Hugs を使う

Haskell を使うには，標準的な処理系が 2 つある．

• Hugs: Haskell Users Gofer System（http://www.haskell.org/hugs/）
• GHC: Glasgow Haskell Compiler（http://www.haskell.org/ghc/）

試してみるには解釈系が楽なので Hugs を使おう．Hugs ではまず前もってエディタを使いスクリプトを作り，次に Hugs を呼び出してセッションを始める．スクリプトのファイル名には Haskell script の意味で .hs をつける．簡単な階乗のスクリプトの例をファイルを cat して眺める（コメントは {-, -} の間，または -- と行末の間に書く）．

```
% cat factorial.hs
factorial         :: Integer -> Integer
    {-factorial の型は Integer をもらい Integer を返す -}
factorial n       = if n == 0 then 1
                              else n * factorial (n - 1)
```

次に Hugs を起動すると以下のような ASCII アート状の画面がでる．いまのプログラムを実行してみる．

```
% hugs

__   __ __  __  ____   ___      ________________________________________
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2003
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                        Report bugs to: hugs-bugs@haskell.org
||   || Version: November 2003 ________________________________________

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Prelude> :l factorial.hs
Main> factorial 10
3628800
Main> :q
[Leaving Hugs]
%
```

Prelude> は基本関数を定義している Prelude というモジュールの動作中を示す．:l factorial.hs はスクリプト factorial.hs をロードする指令．:q は Hugs 終了の指令である．

関数プログラミングの復習

最初に関数プログラミングをさらっと復習する．ある機能は Haskell だけのものである．

• 再帰呼出し

これはほとんど説明を要しない（手続き型言語でもできるが）．再帰呼出しは基本中の基本である．例はクイックソートにすでにあった．

• 第一級身分の関数 first class status function

関数を引数として関数に渡せる．また関数を結果として関数から返せる．関数プログラミングなので当然か．Lisp 系の言語でもでき，Scheme なら関数を引数に渡す例は map で，map は第 1 引数に関数を，第 2 引数にリストをもらい，リストの各要素に関数を作用させた結果のリストを返す．

```
> (map factorial '(1 0 2 3))
(1 1 2 6)
```

一方，Haskell ではいまの階乗の関数を利用し

```
Prelude> :l factorial.hs
Main> map factorial [1,0,2,3]
[1,1,2,6]
```

関数を返す例は Curry 化である．add のような通常 2 引数をとる関数は add 1 3 のように使う．このとき add 1 までで次に受け取る引数に 1 を足す関数が返ると解釈する．add 1 3 では返ってきた関数を 3 に作用させ，結

果は 4 になる (だから (add 1) 3 なのだが関数は左に結合するのでかっこは不要).

したがって add は「引数 1(Integer) をもらい，(3(Integer) をもらって 4(Ingeter) を返す) 関数を返す」という意味で Integer ->(Integer -> Integer) の型と考える．-> は右に結合するとして，通常はこのかっこを省き，add の型を Interger -> Ingetger ->Integer と表記する．このように 2 引数の関数を，1 引数をもらい 1 引数の関数を返すと考えることを Curry 化 (currying, Curry 化された関数は curried function) という．Haskell はこの機構を考えた Haskell B. Curry のファーストネームである．

- 多相型 polymorphic typing

似たような型のクラスに対し，演算を決めることができる．Integer も Float も Num 型クラスで，そのクラスのいずれの型にも乗算 * が使えるように一括で定義できる．

```
(*) :: Num a => a -> a -> a
```

- リストの内包表記 list comprehension

これもクイックソートに登場した．別の例では

```
Prelude> [[x,y]|x<-[1..3],y<-[2..4]]
[[1,2],[1,3],[1,4],[2,2],[2,3],[2,4],[3,2],[3,3],[3,4]]
```

y を 2, 3, 4 と変えながら x を 1, 2, 3 と変え，[x,y] のリストを作る．

- 引数のパターンマッチ

同じくクイックソートにあった．引数が [] のときの定義とそれ以外 x:xs のときの定義が分けて書いてあった．さらにこの場合は引数のリストの car を x, cdr を xs として定義の本体で使えるのが嬉しい．次の例は引数を 0 と 0 以外に分けて定義している．

```
factorial 0 = 1
factorial n  = n * factorial (n - 1)
```

これは

```
factorial n
 |n == 0    = 1
 |otherwise = n * factorial (n - 1)
```

のようにガードを使って書いてもよい．

- 遅延評価 lazy evaluation

Lisp では関数を引数に作用させるとき，まず引数をすべて評価してからそれを関数に渡す．いわゆる作用的順序 (applicable-order) である．評価しないようなものは特殊形式といって例外扱いである．したがって

```
> ((lambda (x y) x) 1 (/ 1 0))
```

は y は使わないにもかかわらず 0 で割ったと叱られる．

一方，必要なときにだけ引数を評価するのを遅延評価とか lazy evaluation といい，Haskell はその陣営にいる．[0..] は 0, 1, 2,... と無限のリストを返すが，

```
Prelude> take 4 [0..]
```

の例では，take 4 が先頭から 4 個だけを要求するから，4 から先は計算せず，結果は

```
[0,1,2,3]
```

となる．

- 副作用対応

関数プログラミングの最大の泣きどころは入出力を含む副作用である．式の中の部品の式はどの順に評価してもよいというのが特徴だったが，そういう部品の式に出力指令があると，評価順に出てくるから，滅茶苦茶になってしまう．Haskell はシリアルに計算を進めるメカニズムの monad を考案し，これを解決したといっているが，関数型を一部手放したというべきであろう．Lisp にも最初から prog があった．

- 2 項演算子と関数名

Lisp 系言語の関数呼出し，関数作用ははすべて演算子をリストの先頭に書く．(+ 1 2 3 4) も (plus 1 2 3

4）も区別はない．Huskell では 2 項演算子も自由に使える．ただし 2 項演算子は + や ++ のように特殊記号だけで構成し，被演算子の間におく．したがって優先順位を決める必要がある．関数作用より弱い．

一方演算子（というか関数名）を先頭におく方法ももちろんあり，関数名は英字が主力になっている．しかし 2 項演算子を関数名として使う方法や関数名を 2 項演算子とする方法も用意してある．

```
Prelude> 1 + 3
4
Prelude> (+) 1 3      {- 2 項演算子をかっこで囲むと関数名になる -}
4
Prelude> min 1 3
1
Prelude> 1 `min` 3    {- 関数名をバッククォートで囲むと 2 項演算子になる -}
1
```

- λ 式

```
primes = sieve [2..]
  where sieve (s:ss) = s : sieve (filter (\x -> x `mod` s > 0) ss)
```

のスクリプトで 25 個の素数を計算すると

```
Main> take 25 primes
[2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71,73,79,83,89,97]
```

`\x -> ` が $\lambda x.$ に相当する．`[2..]` つまり 2, 3, 4... に `sieve` を作用させると素数列 `primes` が得られる．`sieve` はもらってきた引数 `s:ss` の先頭 `s` と，（残り `ss` を `Bool` 型の関数（`\x -> x `mod` s > 0`）でフィルタし，再び `sieve` にかけたもの）を `cons` すると読む．`Bool` 型の関数の方は `ss` から順にもらった要素を `x` とし，`x` を `s` で割った剰余 `> 0` と読む．

車両のソート問題

お話だけでは面白くないので，例題をやってみよう．Donald Knuth の The Art of Computer Programming, 第 3 巻の演習問題 5.2.4-19 である（図 -1 左）．

図 -1 右を見て欲しい．同じような図が 8 つある．左上から右へ A, B, C, D. 左下から右へ E, F, G, H とする．これは図左のように 1 番線から 3 番線まで行き止まりの線が U ターン線で接続されているところを示す．0 番線に 0 から 7 まで番号のついた車両がいる．路面電車と思えばよい．3 本の線をスタックとして使い，3 番線の左の出口から 0 号車から順に車両を出すにはどうするか．そのプログラムを書くのである．

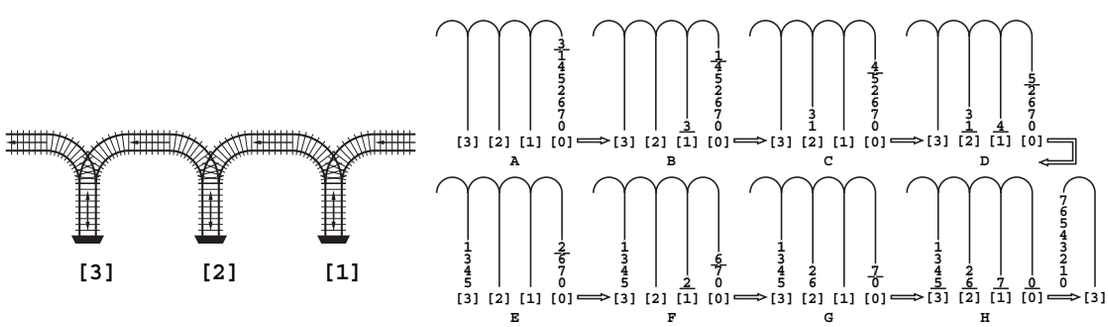

図 -1

一般に n 番線まであれば，2^n 両をソートすることができる．図 -1(E) は上半分の 4 両が 3 番線に昇順に収まったところである．昇順というのはここから発車する順が増加するという意味である．

上段と下段は上半分の 4 両と下半分の 4 両について同じように処理している．ところどころ車両の下に横線が引い

てあるが，その上を処理するという意味だ．

まず 0 番線の 3 号車を 1 番線に入れる (B)．1 番線と 0 番線の横線の上は昇順になっていると思い，昇順にマージして 2 番線に入れる．U ターンするので 2 番線には降順に収まる (C)．続いて 0 番線の 4 を 1 番線に入れる (D)．2, 1, 0 番線に降順に車両がいるから，降順にマージして 3 番線に入れる (E)．同様にして H では 3, 2, 1, 0 番線に昇順に車両が入った．そこで昇順にマージしながら出口から発車させる．

プログラムは以下のようになった．

```
decMergen, incMergen    :: [[Int]] -> [Int]      -- 整数のリストのリストをもらいリストを返す
decMergen (xs0:xs1:[])  = decMerge2 xs0 xs1      -- リストが 2 つのとき マージ
decMergen (xs0:xs1:xss) = decMergen ((decMerge2 xs0 xs1):xss)    -- 3 つ以上のとき
incMergen (xs0:xs1:[])  = incMerge2 xs0 xs1
incMergen (xs0:xs1:xss) = incMergen ((incMerge2 xs0 xs1):xss)

decMerge2, incMerge2    :: [Int]->[Int]->[Int]   -- 2 つのリストのマージ Curry 化
decMerge2 [] ys         = ys                     -- 前のリストが終った
decMerge2 (x:xs) []     = x:xs                   -- 後のリストが終った
decMerge2 (x:xs)(y:ys)
 | x>y                  = x:(decMerge2 xs (y:ys))      -- 小さいほうを出力へ
 | otherwise            = y:(decMerge2 (x:xs) ys)
incMerge2 [] ys         = ys
incMerge2 (x:xs) []     = x:xs
incMerge2 (x:xs)(y:ys)
 | x<y                  = x:(incMerge2 xs (y:ys))
 | otherwise            = y:(incMerge2 (x:xs) ys)

decSort, incSort :: Int -> [Int] -> [Int]       -- データ長 ソートするリスト 結果リスト
decSort n cs
 | n == 2    = decMergen ([head cs]:[tail cs])  -- 2 つのとき リストにしてマージ
 | otherwise = decMergen (xs:ys)                -- 4 つ以上のとき 途中で半分に
                 where xs = reverse (incSort n2 (take n2 cs))    -- 前半をソート
                       ys = decMove n2 (drop n2 cs)              -- 後半を分配
                       n2 = n `div` 2
incSort n cs
 | n == 2    = incMergen ([head cs]:[tail cs])
 | otherwise = incMergen (xs:ys)
                 where xs = reverse (decSort n2 (take n2 cs))
                       ys = incMove n2 (drop n2 cs)
                       n2 = n `div` 2

decMove, incMove  :: Int -> [Int] -> [[Int]]    -- スタックに分配する
decMove n cs
 | n == 2    = [head cs]:[tail cs]
 | otherwise = xs:ys
                 where xs = decSort n2 (take n2 cs)      -- 途中で半分に 前半ソート
                       ys = decMove n2 (drop n2 cs)      -- 後半 分配
                       n2 = n `div` 2
incMove n cs
 | n == 2    = [head cs]:[tail cs]
 | otherwise = xs:ys
                 where xs = incSort n2 (take n2 cs)
                       ys = incMove n2 (drop n2 cs)
                       n2 = n `div` 2

cars8 :: [Int]              -- テスト用データ
cars8 = [3,1,4,5,2,6,7,0]
```

このプログラムはもちろん n が 2 の冪乗でないとうまくいかない．Int は有限の整数の型，drop n は take n

の逆でリストの先頭から n 個を削除する．昇順と降順の両方が出てくるので，マージもソートも incMerge とか decSort のように両用の構えに作ってある．c を inc か dec として cMergen は任意の本数のソート済みのリストを 1 本にマージする．cMergen の実働部分は 2 本のマージで cMerge2 で定義している．

cSort と cMove の最初の引数は対象の車両数を示す．たとえば 8 両の cSort は 4 両を cSort し，U ターンのため reverse をかけ（3 番線に入れる），（2 番線以下に）残りの 4 両をおく cMove 4 の結果に：で cons し，それを cMergen している．

だんだんと n が半減し，2 になると別の処理をしている．定義を条件付きに書くには，前にも例があったが縦棒でガードを作る．実行結果は以下の通り．

```
Prelude> :l railstack.hs
Main> incSort 8 cars8
[0,1,2,3,4,5,6,7]
Main> decSort 8 cars8
[7,6,5,4,3,2,1,0]
```

亦楽しからず乎．

参考文献

1) http://www.cis.upenn.edu/proj/plclub/contest/index.php
2) 原文 http://www.md.chalmers.se/~rjmh/Papers/whyfp.html
   和訳 http://www.sampou.org/haskell/article/whyfp.html
3) 原文 http://www.haskell.org/tutorial/
   和訳 http://www.sampou.org/haskell/tutorial-j/index.html
4) http://www.willamette.edu/~fruehr/haskell/evolution.html
5) Backus, J.: Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, CACM, Vol.21, No.8, pp.613-641.
   www.stanford.edu/class/cs242/readings/backus.pdfでも見ることができる．
   翻訳はbit, Vol.11, No.9, pp.14-29, No.10, pp.19-32, No.11, pp.52-61, また 赤摂也編: ACMチューリング賞講演集, 共立出版 (1989). 米澤明憲による原著の紹介は会誌43巻9号の名著名論にある．
6) Milner, R., Tofte, M. and Harper, R.: The Definition of Standard ML, The MIT Press (1990).
   Milner, R. and Tofte, M.: Commentary on Standard ML, The MIT Press (1991).
7) Turner, D. A.: Miranda: A Non Strict Functional Language with Polymorphic Types, in Functional Programming Language and Computer Architecture, LNCS 201 (Sep. 1985).
8) Jones, S. P. ed.: Haskell 98 Language and Libraries, The Revised Report, Cambridge University Press (2003).
   原文 http://www.haskell.org/definition/haskell98-report.pdf
   和訳 http://www.sampou.org/haskell/report-revised-j/index.html

（平成17年2月18日受付）