■ 連載

Haskell プログラミング

木（tree）で遊ぶ

山下 伸夫（（株）タイムインターメディア）
nobsun@sampou.org

プログラムを読もう

どのような言語を使うにせよ，プログラミングを楽しむためにはその言語の構文と文法を身につけなければならない．これは自然言語と同じである．しかし，エッセンスのみ取り出した構文と文法だけを暗記するような学習はあまり楽しいものではない．今回は，Haskell を用いた簡単なプログラミングを示し，そのプログラムコードの「読み方」をできるだけ丁寧に説明する．アルゴリズム自体はさほど難しいものではないので，Haskell で記述するとどのようになるか理解していただけると思う．プログラムコードが読めるようになれば，書くのはすぐにできるようになるだろう．

例題 1：二分木の列挙

「型慣らし」にごくやさしい問題をやってみよう．

課題　末端のノードが n 個の二分木を列挙せよ．

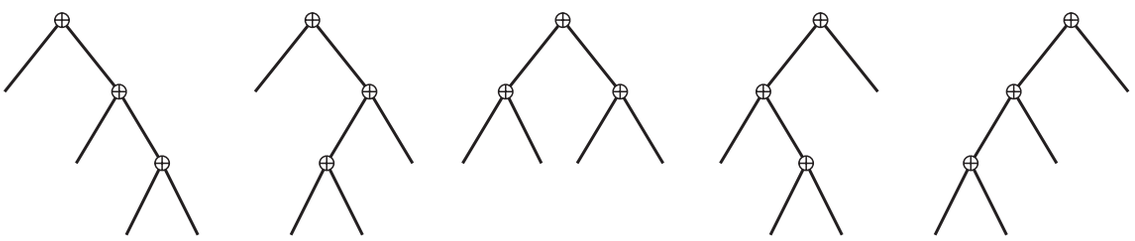

図 -1　末端が 4 個の二分木

たとえば，末端が 4 個の二分木は図 -1 のように 5 個ある．これを末端が n 個の場合に一般化しようというのである．このような二分木を Haskell ではどのように定義するのだろうか．Haskell では組み込みの型以外をプログラマが data 宣言によって定義できる．たとえば，このように定義する．

```
data DumbTree = Empty | Fork DumbTree DumbTree
```

これは新しいデータ型を作るときの Haskell の構文で data 宣言といい，「DumbTree 型は，末端ノード Empty かあるいは（再帰的に）2 つの部分木 DumbTree を含む Fork ノードで構成される」と読む．このようなデータ宣言を

読む場合には，DumbTree は型構成子 (type constructor)，Empty および Fork はデータ構成子 (data constructor) であることに注意する．型構成子を適用して型を得ることと，データ構成子を適用して値を得ることは別である．前者はコンパイルの仕事で，プログラム全体にわたっての型の整合性をチェックするときのみ用いられ，後者は実行の仕事で実際の計算法を決めるものである．

```
-- trees 整数(>0)から，DumbTreeのリストへ
trees :: Int -> [DumbTree]
trees 1 = [Empty]
trees n = concat [ joins ls rs | (ls,rs) <- [ lrs xs ys | (xs,ys) <- splits1 n ]]

-- splits1 整数(>0)から，和がその数になる整数(>0)の2つ組のリストへ
splits1 :: Int -> [(Int,Int)]
splits1 1 = []
splits1 n = (1,n-1) : [ (i+1,j) | (i,j) <- splits1 (n-1) ]

-- lrs 2つの整数から，それぞれの大きさのDumbTreeのリストの2つ組へ
lrs :: Int -> Int -> ([DumbTree],[DumbTree])
lrs xs ys = (trees xs, trees ys)

-- joins 2つのDumbTreeのリストから，それらをそれぞれ左右に持つDumbTreeのリストへ
joins :: [DumbTree] -> [DumbTree] -> [DumbTree]
joins ls rs = [ Fork l r | l <- ls, r <- rs ]
```

trees が与えられた整数 (Int は Haskell に組み込みの固定長整数) を二分木に変換する関数である．最初の行は，trees がどのような型を持っているかの型宣言で，「trees は Int 型から DumbTree 型を要素とするリストへの関数という型を持つ」と読む．型宣言中の [T] は T 型を要素とするリストという型を示す．Haskell のリスト型は要素の型が単一でなければならないことに注意する．2 行目は自明な場合の定義である．1 個の末端を持つ木は Empty ノード 1 つだけの木が唯一である．それゆえ，それだけを含むリスト [Empty] である．こちらの [Empty] は「型」ではなく「データ」である．3 行目は 1 より大きい n に trees を適用したものの定義である．その右辺は，

```
concat 𝒳
```

というかたちをしている．これの $\mathcal{X}$ の部分は，

```
[ joins ls rs | (ls,rs) <- 𝒴 ]
```

というかたちである．これはリストの内包表記と呼ばれる構文である．リストの内包表記 [f x | x <- xs] は数学の $\{ f(x) \mid x \in X \}$ と対応しており，Haskell での意味は，map f xs と同等である．$\mathcal{Y}$ の部分はリストで，その要素は 2 つ組 (tuple) であることが分かる．このリストの内包表記は，「$\mathcal{Y}$ の要素である 2 つ組のそれぞれ第 1 要素 (ls) と第 2 要素 (rs) に関数 joins を適用した結果構成されるリスト」と読む．$\mathcal{Y}$ の部分は，

```
[ lrs xs ys | (xs,ys) <- splits1 n ]
```

である．「splits1 n で構成されるのはリストの要素である 2 つ組のそれぞれの前の要素 xs と後の要素 ys に関数 lrs を適用した結果構成されるリスト」と読む．

これで trees の読み方が分かったので，今度は内側から見ていこう．splits1 は与えられた整数を左右の部分木に含まれる末端ノードの数 (1 以上) に二分割する関数である．この関数は，Int から Int の 2 つ組のリストへの関数という型を持つ．1 は 1 以上の部分に分割できないので，空リスト []．1 より大きい n に対しては，1 に splits1 を適用してできた 2 つ組のそれぞれについて最初の要素に 1 を加えたリスト (ここまでがリストの内包表記で定義されている) の先頭に，(1,n-1) という 2 つ組を付けたものである．ここで，: は Haskell では組み込みのリストの構成子であり，a -> [a] -> [a] という型を持つ．

lrs は 2 つの分割された数からそれぞれ左右の部分木となるべき二分木をつくる関数である．この関数の意味はあきらかだろう．

joins は左側の枝になるべき部分木のリストと右側の枝になるべき部分木を組み合わせて Fork ノードを構成する関数である．このリストの内包表記は数学の表記法 $\{ App(l,r) \mid l \in L \wedge r \in R \}$ に対応している．これも意味はあきらかだろう．この式で構成されるリストの長さは，引数 ls の長さと引数 rs の長さの積になることに注意する．

最後に concat は標準プレリュードで定義されている関数で，要素がリストであるようなリストに適用すると，要素であるリストを連結し全体として 1 つのリストにする関数である．concat の型は [[a]] -> [a] すなわち a 型の値を要素とするリストのリストから a 型の値を要素とするリストへの関数の型を持つ．ここで，型が分かると関数の意味がよりよく理解できることに注目してもらいたい．Haskell のプログラミングにおいては型宣言は必ずしも必要ではないが，型宣言は関数の仕様のサマリでありプログラマの意図 (の一部) を表現したものと見なせる．それゆえ，型宣言を書いておくことは重要である．

さて，ここまでのコード (DumbTree，trees，splits1，lrs，および joins の定義) をファイル first.hs に保存し，解釈系を使って評価してみよう．今回は解釈系に GHC に付属してくる ghci を使う．シェルからファイルを指定して ghci を起動すると解釈系がファイルをロードしてユーザの入力待ちプロンプトが出る．

```
% ghci first.hs
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 6.4, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base ... linking ... done.
Compiling Main             ( first.hs, interpreted )
Ok, modules loaded: Main
*Main> :quit
%
```

解釈系を終了するには，解釈系のプロンプトから :quit と入力すればよい．では，定義した式を試してみよう．

```
*Main> trees 4

<interactive>:1:
     No instance for (Show DumbTree)
       arising from use of `print' at <interactive>:1
     In a 'do' expression: print it
*Main>
```

何かエラーメッセージらしきものが出ている．DumbTree が Show クラスのインスタンスとして宣言されていないので印字ができないと言っているのである．Haskell の解釈系では，ある型のデータを表示するためには，その型が Show クラスのインスタンスであることを宣言しておかなければならない．Haskell の組み込み型については標準プレリュードで Show クラスのインスタンスとして宣言されているが，DumbTree は新たに定義した型なので，ここで Show クラスのインスタンスであることを宣言する．型クラスについては回をあらためて説明する．ここでは，DumbTree 型が Show クラスのインスタンスであることを宣言する具体的な方法のみを確認しておく．これには，DumbTree 型のデータ用の Show 関数を定義する．Show 関数は当該の型のデータからそのデータの印字表現となる文字列への変換関数である．この場合 Show の型は DumbTree -> String である．

```
instance Show DumbTree where
   show Empty      = "0"
   show (Fork l r) = "(" ++ show l ++ "^" ++ show r ++ ")"
```

末端ノード Empty のときは文字列 "0" に，Fork ノードのときは，まずカッコ ("(")，左の枝の印字表現，文字列 "^"，そして右の枝の印字表現，最後にコッカ (")") を，連結したものである．++ はリストを連結する演算子である．Haskell において文字列型 String は文字型のリスト [Char] の別名である．したがって，文字列はリストを連結する演算子 ++ を用いて連結することができる．このインスタンス宣言を先刻の first.hs ファイルに追加しよ

う．そうしておいて，ghci の対話環境の中からこれを :load コマンドを使ってロードし，もう一度，trees 4 を評価してみよう．

```
*Main> :load first.hs
Compiling Main              ( first.hs, iterpreted )
Ok, modules loaded: Main.
*Main> trees 4
[(0^(0^(0^0))),(0^((0^0)^0)),((0^0)^(0^0)),((0^(0^0))^0),(((0^0)^0)^0)]
*Main>
```

これは，図 -1 に対応しているのが分かるだろうか．

練習問題 1　trees 4 の評価結果が

```
[(L^(L^(L^R))),(L^((L^R)^R)),((L^R)^(L^R)),((L^(L^R))^R),(((L^R)^R)^R)]
```

と表示されるよう DumbTree 用の show の定義を変更せよ．

例題 2: 切符問題

筆者は最近はプリペイドカードなどを使うことが多く，あまり切符を購入しなくなったが，近距離の鉄道の切符には 4 つの数字が印刷されていた．この 4 つの数字と四則を使って，10 を作るという遊びをやったことがあるかもしれない [1]．ここでは以下のような問題を設定する．

課題　与えられた 1 桁の数すべてと四則演算 (+，-，*，/) を使って，指定された数の作り方を示せ．

たとえば，「1，1，9，9 を使い 10 を作れ」なら，(1+(1/9))*9 が解答 (の 1 つ) である．ここでは，与えられる 1 桁の数の個数は限定しない．

四則計算を表す計算木を定義する

この問題を解くのに，与えられた数字で考えられるすべての四則計算式を構成してそれが指定された数になるかをチェックするという方法を採用しよう．計算式は図 -2 のように木で表現できる．この木 (図 -2) の形は先刻の例題にあった木 (図 -1) と同じである．

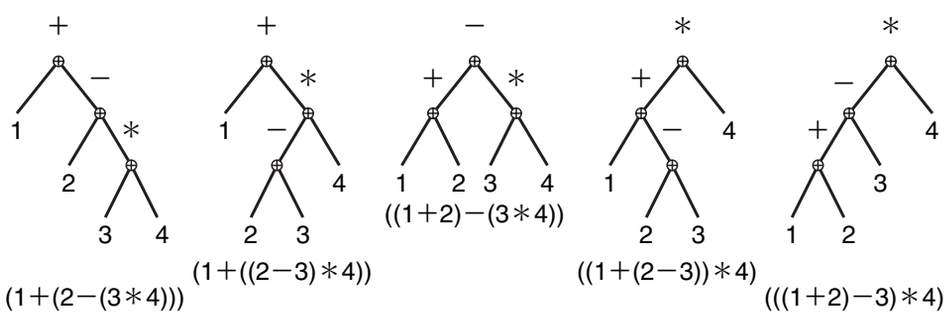

図 -2　計算式と対応する木

先刻の例題では木の形だけを問題にしたので，木には何の付加情報も持たせてはいなかったが，この問題で使う計算木 (Term 型) では，末端ノードには数字 (Char 型) を，節のノードには演算子 (Char 型) を，それぞれ持たせる．Haskell では以下のように定義する．

```
data Term = Val Char | App Char Term Term
```

これは，「Term 型は，Char 型の値を含む Val ノードか，Char 型の値と，(再帰的に) 2 つの部分木 (Term 型) を含む App ノードで構成される」と読む．

リストから計算木へ

先刻の例題では末端ノードの数から木を構成したが，今度は末端ノードに配置される数字のリストと節ノードに配置される演算子のリストから計算木を構成する．

```
trees :: [Char] -> [Char] -> [Term]
trees ds os = [ t | (_,t) <- [ otree os u | u <- dtrees ds] ]
{- (_,t)のアンダースコアは2つ組の最初の項は使用しないことを示す -}

dtrees :: [Char] -> [Term]
dtrees [x] = [Val x]
dtrees ds  = concat [ joins ls rs | (ls,rs) <- [ lrs xs ys | (xs,ys) <- splits1 ds ]]

splits1 :: [Char] -> [([Char],[Char])]
splits1 [x]    = []
splits1 (x:xs) = ([x],xs) : [ (x:ys,zs) | (ys,zs) <- splits1 xs ]

lrs :: [Char] -> [Char] -> ([Term],[Term])
lrs xs ys = (dtrees xs,dtrees ys)

joins :: [Term] -> [Term] -> [Term]
joins ls rs = [ App '^' l r | l <- ls, r <- rs ]

otree :: [Char] -> Term -> ([Char],Term)
otree os (Val c)     = (os,Val c)
otree os (App _ l r) = (os'', App o' l' r') -- 変数名(先頭文字以外)に ' を使用できる．
    where (o':os',l') = otree os  l          -- 定義の左辺にパターンを使用できる
          (os''  ,r') = otree os' r

instance Show Term where
   show (Val c)     = [c]
   show (App o l r) = "(" ++ show l ++ [o] ++ show r ++ ")"
```

ここでは見通しを良くするために，末端ノードに数字を配置するパス (dtrees) と，節ノードに演算子を配置するパス (otree) に分けてある．dtrees は，先刻の例題の trees と比べると，引数が Int から [Char] に，返り値が [DumbTree] から Term に代わっているが，先刻の例題の trees と同じ構造である．数字のリストを左右に分けるのが splits1，左右の数字のリストから左右の部分木を構成するのが lrs，左右の部分木のリストから演算適用の App ノードを合成するのが joins である．joins では，演算子は仮に '^' に固定してある．

otree が，演算子のリストと，演算子を '^' に固定した計算木から，与えられた演算子を持つ計算木を構成する関数である．otree の定義の 1 行目は例によって，型宣言で，「otree は，Char 型の値を要素とするリスト (演算子のリスト) と Term 型の計算木 (演算子がまだ配置されていない計算木) とから，未配置の演算子のリストと Term 型の計算木 (演算子も配置済みの計算木) の 2 つ組への関数の型を持つ」と読む．返り値が計算木だけではなく，未配置の演算子リストを含んでいることに注意すること．2 行目は，「第 2 引数の計算木が末端ノードなら，第 1 引数の演算子リストと，その末端ノードを 2 つ組にしたもの」と読む．3 行目は，「第 2 引数が演算適用ノードなら，os'' と，演算子を o' とし，左右の部分木を l' および r' とする演算適用ノードとする」と読む．次の行の where 節 (where clause) は局所定義である．ここで，o' は otree を元の計算木の App ノードの左部分木 l に適用した結果の 2 つ組の第 1 要素のリストの先頭．l' はその第 2 要素，r' はその第 1 要素のリストの残りの部分 os' に otree を適用した結果の第 2 要素．そして，os'' はその結果の第 1 要素である．

Term の場合も表示するためには Show クラスのインスタンスにする必要がある．そのインスタンス宣言が最後の 3 行である．ここまでで，計算木の構成部分が定義できたので実際に評価してみよう．Term データの定義以降の関数の定義をファイル second.hs に保存して，ghci にロードし，途中の関数の様子を見よう．

```
*Main> :load second.hs
Compiling Main             ( second.hs, iterpreted )
Ok, modules loaded: Main.
*Main> splits1 "0123"
[("0","123"),("01","23"),("012","3")]
*Main> lrs "01" "23"
([(0^1)],[(2^3)])
*Main> [lrs xs ys |(xs,ys)<- splits1 "0123"]
[([0],[(1^(2^3)),((1^2)^3)]),([(0^1)],[(2^3)]),([(0^(1^2)),((0^1)^2)],[3])]
*Main> [joins ls rs | (ls,rs)<-[lrs xs ys |(xs,ys)<- splits1 "0123"]]
[[(0^(1^(2^3))),(0^((1^2)^3))],[((0^1)^(2^3))],[((0^(1^2))^3),(((0^1)^2)^3)]]
*Main> otree "+-*" (App '^' (App '^' (Val '0') (Val '1')) (App '^' (Val '2') (Val '3')))
("",((0+1)-(2*3)))
*Main> trees "1234" "+-*"
[(1+(2-(3*4))),(1+((2-3)*4)),((1+2)-(3*4)),((1+(2-3))*4),(((1+2)-3)*4)]
*Main>
```

trees "1234" "+-*" を評価すると図-2 に対応する計算木ができたことが確認できる．

計算木を評価する

計算木を評価する関数を定義する．計算は割り算があるので，有理数で行うようにする．まず，有理数を分母と分子の 2 つ組として定義する．

```
type Rat = (Int,Int)
```

この type 宣言は，既存の型に別名を付けるものである．計算木の評価関数 eval は以下のとおりである．

```
eval :: Term -> Rat
eval (Val x)     = ctor x
eval (App o l r) = (ctoo o) (eval l) (eval r)

ctor :: Char -> Rat
ctor x = (ord x - ord '0',1)

ctoo :: Char -> (Rat -> Rat -> Rat)
ctoo '+' (x,y) (z,w) = (x*w+z*y,y*w)
ctoo '-' (x,y) (z,w) = (x*w-z*y,y*w)
ctoo '*' (x,y) (z,w) = (x*z,y*w)
ctoo '/' (x,y) (z,w) = if z == 0 then (0,0) else (x*w,y*z)
```

eval の定義はやさしいだろう．その型は「計算木 Term 型から有理数 Rat 型への関数の型」である．Val ノードのときは含まれている数字を有理数に変換したもの．App ノードのときは，演算子文字 o に ctoo を適用して得た Rat 同士の演算関数を，左の部分木を評価した結果と右の部分木を評価した結果に適用する．

ctor は 1 桁の整数を表現する Char 型のデータを有理数 Rat 型に変換する．ここで使われている ord は文字の ASCII コードを返す関数で，標準プレリュードでは定義されておらず，標準ライブラリの Char モジュールで定義されている．これを利用するにはソースコードファイルですべてのデータや関数の定義の前に import 宣言によって Char モジュールをインポートする．

```
import Char
```

この 1 行を second.hs の先頭に追加しておこう．

可能なすべての計算木の構成

残りは可能な計算木をすべて構成して，そのなかで指定された数になるものを選べばよい．

```
ticket :: Int -> [Char] -> Term
ticket n ds = head (filter (same n) (allterms ds))

same :: Int -> (Term -> Bool)
same i t = i*d == n && d /= 0
   where (n,d) = eval t

allterms :: [Char] -> [Term]
allterms ds = concat [ trees ns os | ns <- perm ds, os <- rperm ops4 (length ds - 1) ]

ops4 = "+-*/"

perm [] = [[]]
perm xs = concat [ pm hs ts | (hs,ts) <- splits xs ]
   where pm _  []     = []
         pm hs (t:ts) = [ t:ys | ys <- perm (hs ++ ts) ]

splits []     = [([],[])]
splits (x:xs) = ([],x:xs) : [ (x:ys,zs) | (ys,zs) <- splits xs ]

rperm _  0 = [[]]
rperm [] _ = []
rperm xs n = [ x:ys | x <-xs, ys <- rperm xs (n-1) ]
```

ticket が作るべき数 n を与えられた数字のリスト ds から四則を使って構成し表示する関数である．allterms で可能なすべての計算木を構成しその中から評価すると n と等しい有理数になるものだけを濾過 (filter) し，それが空リストなら構成できなかった旨の文字列を，空ではないリストなら，その先頭を文字列に変換したものとなる．perm は与えられた数字のリストの順列を構成する関数，rperm は与えられた演算子を重複を許して n 個使う順列を構成する．これらの定義を second.hs に追加して ghci にロードすると実行できる．

```
*Main> :load second.hs
Compiling Main             ( second.hs, iterpreted )
Ok, modules loaded: Main.
*Main> ticket 10 "1158"
(8/(1-(1/5)))
*Main> ticket 24 "1346"
(6/(1-(3/4)))
*Main>
```

これで切符問題を解くプログラムが書けた．Haskell を使ったプログラミングではデータや関数の在り様を記述するだけでよいという雰囲気を味わってもらえただろうか．

今回のコードは説明しやすいように書いたので，実際に利用するプログラムとしては改良すべき点がいくつかある．これらの一部を練習問題としておく．

練習問題 2　ticket 関数は，解が存在しないときにエラーになる．これを変更して解が存在しないときはその旨のメッセージを表示するようにせよ．

練習問題 3　与えられた数字に重複があるときには，perm が同じ並びを複数回構成する．このため，計算木を重複してチェックすることになる．数字の順列の構成部分を改良し，計算木のチェックの重複を除去せよ．

練習問題 4　2 パスになっている trees を 1 パスにせよ．すなわち，数字のリストから構成した仮の計算木をたどりながら，最終的な計算木を構成し直すのではなく，一度で演算子も割り当てた計算木を構成するように改造せよ．

練習問題 5　計算木が構成されると同時にその計算木の評価もなされるように，上で改造した trees に評価のパスも組み込め．

注）連載「Haskell プログラミング」に掲載されたプログラムは次の URL から取ることができる．
http://www.sampou.org/haskell/ipsj/

参考文献
1）中西正和: 数楽オリンピックへのいじわる，情報処理，Vol.11, No.7, pp.426-427 (July 1970).

（平成17年3月16日受付）