第五回Haskell読書会

第6章　再帰関数

ふるた　たけし

6.1 基本概念

- 階乗（n!）
  - ☑ ライブラリ関数の定義

```
factorial :: Int -> Int
factorial  n = product[1..n]
```

  - ☑ 再帰の定義

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

  - ☑ 例

```
factorial 3 = 3 * factorial 2
            = 3 * (2 * factorial 1)
            = 3 * (2 * (1 * factorial 0))
            = 3 * (2 * (1 * (1))) = 6
```

6.1 基本概念

- 階乗（n!）
  - ☑ ライブラリ関数の定義

    factorial :: Int -> Int

    factorial n = product[1..n]
  - ☑ 再帰の定義

    factorial 0   = 1

    *factorial (n+1) = (n+1) * factorial n*
  - ☑ 例

    *factorial 3 = 3 * factorial 2*

    *= 3 * (2 * factorial 1)*

    *= 3 * (2 * (1 * factorial 0))*

    *= 3 * (2 * (1 * (1))) = 6*

ポイント

- 関数は自分自身を使って定義
- 引数の値が1つずつ減っていくので無限ループにならない
- 引数の値が０の時にストップ
- ストップするところに乗法の単位元１を定義
- ライブラリ関数の方が簡潔であるが、再帰を使った方が分かりやすい
- 数学的帰納法で説明しやすい

掛け算の定義

- 掛け算の定義

  m * n = m + m + … + m　　（m を ｎ回 足す）

- 再帰を使って定義
  - ☑ 定義

    (*) :: Int -> Int -> Int

    *m * 0　　= 0*

    *m * (n +1) = m + (m * n)*
  - ☑ 例

    *4 * 3 = 4 + (4 * 2)*

    *= 4 + (4 + (4 * 1))*

    *= 4 + (4 + (4 + (4 * 0)))*

    = 4 + 4 + 4 + 0

    = 4 + 4 + 4 = 12

6.2 リストに対する再帰

- 再帰は数値だけではなくリストに対しても可能
- ライブラリ関数productを再帰を使って定義
  - ☑ 定義

```
product :: Num a => [a] -> a
product [] = 1
product (n:ns) = n * product ns
```

  - ☑例

```
product [2,3,4] = 2 * product[3,4]
                = 2 * (3 * product[4])
                = 2 * (3 * (4 * product[]))
                = 2 * (3 * (4 * 1))) = 24
```

- リストは*cons*演算子を使った合成データ

```
[2,3,4] = 2:(3:(4:[]))
```

ライブラリ関数lengthの定義

- リストの長さ（要素の数）を計算する関数の定義
  - ☑ 定義

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

  - ☑ 空要素の場合の長さは０
  - ☑ 残りの要素の数に１を加える
  - ☑ 要素の中にワイルドカードを使っている。

ライブラリ関数reverseの定義

- リストの中身を逆にする関数の定義
  - ☑ 定義

```
reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = reverse xs ++ [x]
```

  - ☑ 例

```
reverse[1,2,3] = reverse[2,3] ++ [1]
               = (reverse[3] ++ [2]) ++ [1]
               = ((reverse[] ++ [3]) ++ [2]) ++ [1]
               = (([]++ [3]) ++ [2]) ++ [1]
               = [3,2,1]
```

リスト連結演算子++の定義

- リストを連結する演算子++も再帰で定義可能
  - ☑ 定義

```
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x :xs) ++ ys = x : (xs ++ ys)
```

  - ☑ 例

```
[1,2,3] ++ [4,5] = 1:([2,3] ++ [4,5])
                 = 1:(2:([3] ++ [4,5]))
                 = 1:(2:(3:([] ++ [4,5])))
                 = 1:(2:(3:([4,5])))
                 = [1,2,3,4,5]
```

  - ☑ 後ろのリストは変化させず、前のリストを分解して行く

値を挿入する関数

- insert関数
  - ☑ 定義

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) | x <= y     = x : y : ys
                | otherwise = y : insert x ys
```

  - ☑ 例

```
insert 3 [1,2,4,5] = 1 : insert 3 [2,4,5]
                   = 1 : 2 : insert 3 [4,5]
                   = 1 : 2 : 3 : [4,5]
                   = [1,2,3,4,5]
```

応用：挿入ソート

- isort関数を定義
  - ☑ 定義

```
isort :: Ord a=> [a] -> [a]
isort [] = []
isort (x : xs) = insert x (isort xs)
```

  - ☑ 例

```
isort[3,2,1,4] = insert 3 [2,1,4]
               = insert 3 (insert 2 (insert 1 (insert 4 [])))
               = insert 3 (insert 2 (insert 1 [4]))
               = insert 3 (insert 2 [1,4])
               = insert 3 [1,2,4]
               = [1,2,3,4]
```

6.3 複数の引数

- 同時に複数の引数を変化させる再帰
- ライブラリ関数zipの定義
  - ☑ 定義

```
zip :: [a] -> [b] -> [(a, b)]
zip [] _ = []
zip _ [] = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

  - ☑ 例

```
zip ['a','b','c'] [1,2,3,4] = ('a', 1) : zip ['b','c'][2,3,4]
                            = ('a', 1) : ('b', 2) : zip ['c'][3,4]
                            = ('a', 1) : ('b', 2) : ('c', 3) : zip [][4]
                            = ('a', 1) : ('b', 2) : ('c', 3) : []
                            = [('a', 1) , ('b', 2) , ('c', 3) ]
```

- 注意！　基底部が*2*つ必要

drop関数の定義

- 指定した値でリストを消していくdrop関数の定義
  - ☑ 定義

```
drop :: Int -> [a] -> [a]
drop 0 xs = xs
drop (n+1) [] = []
drop (n+1) (x:xs) = drop n xs
```

  - ☑ 例

```
drop 3 [1,2,3,4,5] = drop 2 [2,3,4,5]
                   = drop 1 [3,4,5]
                   = drop 0 [4,5]
                   = [4,5]
```

6.4 多重再帰

- 自分自身を複数参照する再帰を多重再帰という
- フィボナッチ数列(1,1,2,3,5,8,…)の定義
  - ☑ 定義

```
fibonacci :: Int -> Int
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci (n + 2) = fibonacci n + fibonacci (n + 1)
```

  - ☑ 例

```
fibonacci 4 = fibonacci 2 + fibonacci 3
            = (fibonacci 0 + fibonacci 1)
                    + (fibonacci 1 + fibonacci 2)
              = (0 + 1) + (1 + (fibonacci 0 + fibonacci 1))
            = (0 + 1) + (1 + (0 + 1))
            = 3
```

クイックソートqsortの定義

- クイックソートも多重再帰で定義できる
  - ☑ 定義

```
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x : xs) = qsort smaller ++ [x] ++ qsort larger
                 where
                   smaller = [a| a <- xs, a<=x]
                   larger = [b|b <- xs, b > x]
```

おまけ

- 疑似乱数のなかでよく使われる線形合同法を再帰で定義
- 線形合同法

$R_0=(A \times seed + C) \bmod M$

$R_1 =(A \times R_0 + C) \bmod M$

:

$R_{n+1} =(A \times R_n + C) \bmod M$

☑ 定義（出典：結城浩のHaskell日記）

```
random :: Integer -> Integer -> [Integer]
random seed mx = next seed : random (next seed) mx
        where
            next n = (1812433253 * n) `mod` mx
```

6.5 相互再帰

- ２つ以上の再帰関数をお互いを参照しあって利用
- 偶数（even)・奇数（odd）関数を定義して偶奇判断
  - ☑ 定義

```
even :: Int -> Bool
even 0 = True
even (n + 1) = odd n
odd :: Int -> Bool
odd 0 = False
odd (n + 1) = even n
```

  - ☑ 例

```
even 4 = odd 3 = even 2 = odd 1 = even 0 = True
odd 4 = even 3 = odd 2 = even 1 = odd 0 = False
```

リストへの拡張

- 偶数・奇数関数をリストの偶数・奇数位置のリストに拡張
  - ☑ 定義

```
evens :: [a] -> [a]
evens [] = []
eves (x:xs) = x: odds xs
odds :: [a] -> [a]
odds [] = []
odds (_ : xs) = evens xs
```

  - ☑ 例

```
evens "abcde" = 'a' : odds "bcde"
              = 'a' : evens "cde"
              = 'a' : 'c' : odds "de"
              = 'a' : 'c' : evens "e"
              = 'a' : 'c' : 'e' : odds []
              = 'a' : 'c' : 'e' : []
              = "ace"
```

再帰の秘訣

- 再帰は以下の段階で考える
  - ①　型を定義する
  - ②　場合分けをする。
  - ③　簡単な方を定義する
  - ④　複雑な方を定義する
  - ⑤　一般化し単純にする（高階関数は省略）

例題１：product関数

① 型を定義する

product :: [Int] -> Int

② 場合分けをする。（単位元・基底部を意識）

product [] =

product (n:ns) =

③ 簡単な方を定義する。（単位元・基底部を定義）

product [] = 1

product (n:ns) =

④ 複雑な方を定義する。（再帰で表現できるか意識）

product [] = 1

*product (n:ns) = n * product ns*

⑤ 一般化し単純にする。

定義を整数型から数値型に拡張

product :: Num a => [a] -> a

例題２：drop関数

① 型を定義する

drop :: Int ->[Int] -> [Int]

② 場合分けをする。（単位元・基底部を意識）

drop 0 [] =

drop 0 (x:xs) =

drop (n+1) [] =

drop (n+1) (x: xs) =

③ 簡単な方を定義する。（単位元・基底部を定義）

drop 0 [] = []

drop 0 (x:xs) = x : xs

drop (n+1) [] = []

drop (n+1) (x: xs) =

例題２：drop関数

④ 複雑な方を定義する。（再帰で表現できるか意識）

drop 0 [] = []

drop 0 (x:xs) = x : xs

drop (n+1) [] = []

drop (n+1) (x: xs) = drop n xs

⑤ 一般化し単純にする。

重複をまとめ、使わない値はワイルドカードに置き換え

drop 0 xs = xs

drop (n+1) [] = []

drop (n+1) (_: xs) = drop n xs

例題3：init関数

① 型を定義する

```
init:: [a] -> [a]
```

② 場合分けをする。（単位元・基底部を意識）

```
init (x:xs) =
```

③ 簡単な方を定義する。（単位元・基底部を定義）

```
init(x:xs) | null xs = []
           | otherwise =
```

④ 複雑な方を定義する。（再帰で表現できるか意識）

```
init(x:xs) | null xs = []
           | otherwise = x : init xs
```

⑤ 一般化し単純にする。

```
init :: [a] -> [a]
init [_] = []
init (x:xs) = x: init xs
```

演習問題

1. 演算子の定義
   ① べき乗（ ^ ）を再帰を使って定義
   ② 2^3を簡約化（順番に説明）
2. length[1,2,3],drop3 [1,2,3,4,5],init[1,2,3]を簡約化
3. 標準ライブラリを再帰を使って定義
   ① and ::[Bool] -> Bool
   ② concat::[[a]] ->[a]
   ③ replicate::Int->a->[a]
   ④ (!!)::[a]->Int->a
   ⑤ elem::Eq a => a->[a]->Bool
4. mergeを再帰を使って定義
5. mergeをつかったマージソートmsortの定義
6. 5段階ステップでライブラリ関数の定義
   ① sum
   ② take
   ③ last