課題 3

リストを操作する次の関数をそれぞれ定義しなさい。

(a)

リスト xs の要素に、d が含まれていれば、それを取り除いたリストを作る関数 delete d xs

```
*Main> delete 2 [1,2,3,2,1]
[1,3,2,1]
*Main> delete 5 [1,2,3,2,1]
[1,2,3,2,1]
```

解答及び解説

リストを操作する基本は、空リストの場合と空でない場合に分けること。これには、パターンを使うと分かりやすい。そして、空でないときは、最初の要素と残りに分けて、処理を考える。これもパターンを使うと良い。

さて、本題の要素を取り除くプログラムは次のようになる。

```
delete :: (Eq a) => a -> [a] -> [a]
delete d [] = []
delete d (x:xs) | x == d    = xs
                | otherwise = x:delete d xs
```

空リストの場合、特定の要素は当然含まれていないので、答えは空リスト ([]) である。リストが空でない場合は、最初の要素 (x) が特定の要素 (d) のときと、そうでないときに分かれる。(この場合分けにはガードを使った。) x==d のときは、残り (xs) が答えとなる。x==d でないときは、xs から d を取り除いたものに、x を加えたものが答えとなる。ここで、再帰を用いる。このような操作もリストに対してよく行われる。

(b)

リスト xs の要素に、d が含まれていれば、それをすべて取り除いたリストを作る関数 deleteAll d xs

```
*Main> deleteAll 2 [1,2,3,2,1]
[1,3,1]
```

解答及び解説

(a) とほぼ同じことである。違いは、特定の要素を見つけたときで、すべてを取り除く必要があるため、残りからも特定の要素を取り除かなくてはならず、ここにも再帰が必要であること。

```
deleteAll :: (Eq a) => a -> [a] -> [a]
deleteAll d [] = []
deleteAll d (x:xs) | x == d    = deleteAll d xs
                   | otherwise = x:deleteAll d xs
```

(c)

リスト xs の要素がすべて数値であるとして、その数値の標準偏差を求める関数
deviation xs

```
*Main> deviation [1,2,3]
0.81649...
```

解答及び解説

標準偏差を求める式は、既知のものとしました。そこで、定義通りの計算を用いるならば、次のようになります。

```
deviation xs = sqrt ( (sum (map var xs)) / len )
        where len = length xs
              ave = sum xs / len
              var x = (x - ave)^2
```

実際には、このプログラムは型に整合性がないため実行できません。(型変換を使って実行できるようにすることは可能です。) そこで、プリント No.3 の練習 average のように、関数の再帰定義により求める方法を考えてみます。(プリント No.6 を参照)

関数 average では、要素の合計と個数を計算し、最後に「合計を個数で割る」という求め方をしました。今回は、分散を計算するのに平均値が必要になるため、このままでは同じような計算ができません。

そこで、分散を求める式を少し変形[1] してから計算します。

$$\begin{aligned} x_k\text{の分散} &= \frac{1}{n}\Sigma_{k=1}^{n}(x - \text{平均値})^2 \\ &= \frac{1}{n}\Sigma_{k=1}^{n}x^2 - (\text{平均値})^2 \end{aligned}$$

[1]※ この変形もよく知られたものと思っていましたが、最近はそうでもないようですね。証明は簡単ですので、ここには書かないことにします。

このように変形した式を使えば、まず要素の個数 (n)、要素の和 (sum)、要素の 2 乗の和 (sq) をそれぞれ計算し、それらを使って分散 (var) を、次のように求められます。

$$var \quad = \quad \frac{sq}{n} - \left(\frac{sum}{n}\right)^2$$

従って、プログラムは次のようになります。

```
deviation :: (Floating a) => [a] -> a
deviation xs = deviation' xs 0 0 0

deviation' :: (Floating a) => [a] -> a -> a -> a -> a
deviation' [] n sum sq = sqrt ((sq/n) - (sum/n)^2)
deviation' (x:xs) n sum sq = deviation' xs (n+1) (sum+x) (sq+x*x)
```

ここで、平方根を求める関数 (sqrt) を使うため、数値の型が Floating クラスに限定されています。

(d)

リスト xs が、リスト ys の末尾の切片かどうかを判定する関数 lastSegment xs ys

```
*Main> lastSegment [5, 7] [1, 3, 5, 7]
True
*Main> lastSegment "fghj" "cdfgh"
False
*Main> lastSegment "ff" "fff"
True
```

解答及び解説

リスト ys の先頭からいくつかの要素を取り除いたリストが、xs と同じになれば良い。これを一歩進めると、xs が「ys の先頭の要素を取り除いたもの」の末尾の切片になっていれば、ys の末尾の切片でもある。このことを考えてプログラムを作ると、次のようになる。

```
lastSegment :: (Eq a) => [a] -> [a] -> Bool
lastSegment [] ys = True
lastSegment xs [] = False
lastSegment xs ys@(z:zs) | xs == ys  = True
                         | otherwise = lastSegment xs zs
```

最後の行が、上述の再帰的になっている部分である。その他は、次のような状況を示している。

1. xs が空リストであれば、それはどんなリストに対しても末尾の切片となる

2. ys が空リストであれば、空リスト以外は末尾の切片はない

さて、ここで皆さんにお詫びします。プリント No.3 の練習として、関数 initSegment を出題し、解答をプリント No.7 に掲載しましたが、解答が間違っていました。今回、lastSegment の説明を書いていて気づきました。

改めて、initSegment の正しい定義を掲載しておきます。

```
initSegment :: (Eq a) => [a] -> [a] -> Bool
initSegment [] ys = True
initSegment xs [] = False
initSegment (x:xs) (y:ys) = if x == y then initSegment xs ys else False
```

プリント No.7 のプログラムでは、リスト ys の方が短い場合や、xs と ys が同じリストであるときに答えが求まりません。(前者は False、後者は True が正解。)

(e)

リスト xs の要素がすべて数値であると仮定して、その要素を小さい順に並べたリストを作る関数 seiretsu xs

```
*Main> seiretsu [3, 5, 2, 1, 4]
[1, 2, 3, 4, 5]
```

解答及び解説

いわゆる整列 (sort) の問題です。整列のアルゴリズムは、良く研究されているため様々なものがあります。有名なのは、バブルソート、クイックソート、選択ソートなどでしょう。そして、どのアルゴリズムが効率的かということも良く研究されており、例えばクイックソートはその名の通り、速い (クイック) とされています。

しかし、リストの要素を整列する場合、その構造から演算 (++) の効率が非常に悪いため、この演算を多用するクイックソートは、必ずしも効率が良いとはいえません。

そこで、リストの構造に適したものとして、挿入ソートなどを用いるのが良いでしょう。

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

insert :: (Ord a) => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys) = if x <= y then x:y:ys else y:insert x ys
```

また、挿入ソートはソートアルゴリズムの中ではあまり効率の良い方ではありませんので、もう一つの解答も紹介しておきます。マージソートと呼ばれるものです。これはクイックソートと同程度に効率の良いといわれるアルゴリズムです。しかし、リストの分解と組み立てを繰り返すので、一般的に言われているほどの効率は望めません。

```
seiretsu :: (Ord a) => [a] -> [a]
seiretsu xs = msort (dlist xs)

dlist :: [a] -> [[a]]
dlist [] = []
dlist (x:xs) = [x]:dlist xs

msort :: (Ord a) => [[a]] -> [a]
msort [xs] = xs
msort xss = msort (msort' xss [])

msort' :: (Ord a) => [[a]] -> [[a]] -> [[a]]
msort' [] zss = zss
msort' [xs] zss = xs:zss
msort' (xs:ys:xss) zss = msort' xss (merge xs ys:zss)

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

(f)

リスト xs の要素を、すべての並べ方で並べたリストを要素とするリストを作る関数 narabi xs

```
*Main> narabi [1, 2, 3]
[[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]
```

※ 順番はこの通りでなくて良い。

解答及び解説

これは、すべての順列 (permutation) をリストにする関数である。実は、Haskell には標準関数に同じもの (permutations) があるので、課題を解くのでなければ、それを使うのが良い。

```
import Data.List

narabi xs = permutations xs
```

(注) permutations を利用するには、プログラムの先頭に import Data.List が必要である。

さて、自分で作るとすれば、次のようになる。

```
narabi :: [a] -> [[a]]
narabi [] = [[]]
narabi (x:xs) = foldr1 (++) $ map (insert x) (narabi xs)
    where insert x [] = [[x]]
          insert x (y:ys) = (x:y:ys):(map (y:) (insert x ys))
```

ここでも、効率の悪い演算 (++) を使っているので、何とかしたいところではあるが、残念ながら無理に無くそうとすると、とても分かりにくいものになるか、かえって効率の悪いプログラムになってしまう。

(g)

リスト xs の要素から n 個の要素を選ぶ、すべての選び方で選んだリストを要素とするリストを作る関数 sentaku xs n

```
*Main> sentaku ['a', 'b', 'c'] 2
["ab", "ac", "bc"]
```

※ 順番はこの通りでなくて良い。

解答及び解説

今度は、すべての組み合わせ (combination) をリストにする関数である。残念ながら順列のときのような標準関数 combinations は、用意されていないようである。

さて、次のプログラムのようになる。

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

順列よりも少し簡単である。リスト xs から n 個の要素を選ぶとする。最初の要素を選んだとき、残りから n-1 個を選んで (再帰的) から最初の要素を加えれば良い。但し、n-1 個の選び方が複数あるのため、リストになっている。その一つずつに最初の要素を加えるため、map を使う必要がある。

次に、最初の要素を選ばないときは、残りから n 個を選ぶことになる。これも、再帰的に定義できる。

再帰は、選ぶ個数が 0 個となった場合と選ぶべき要素が無くなった場合に停止する。前者の場合、要素が 0 個のリスト (つまり空リスト) をリストにしたものが答えになる。後者の場合は、選ばれるものが無いので、答えが空リストになる。