高知大学教育学部の情報数学のテキスト
文責　：　高知大学名誉教授　中村　治

Haskell のプログラミング

次に関数型プログラミング言語の例として Haskell のプログラミングを勉強します。インタプリタの GHCI とコンパイラの GHC がありますが、ここではインタプリタの GHCI を使って、一般的な関数型プログラミング言語のプログラミングを練習してみます。Haskell はかなり複雑な言語ですが、興味が出れば最後に挙げてある参考書などで自分で勉強してください。

まずは平均値を求める関数を作ります。$\{x_1, x_2, \cdots, x_n\}$ の平均は

$$\bar{x} = \frac{\sum_{k=1}^{n} x_k}{n}$$

で求められます。リスト $[x_1, x_2, \cdots, x_n]$ の和は $sum[x_1, x_2, \cdots, x_n]$ で与えられ、リスト $[x_1, x_2, \cdots, x_n]$ のサイズは $length[x_1, x_2, \cdots, x_n]$ で与えられるので、プログラムは

```
mean :: [Double] -> Double
mean xs = sum xs / length xs
```

で良いと思って、例えばフリーのサクラエディタを使って、上のプログラムを作り、mean1.hs という名前で適当なフォルダに保存し、コマンドプロンプトをそのフォルダに移動し、

```
ghci mean1
```

とするか

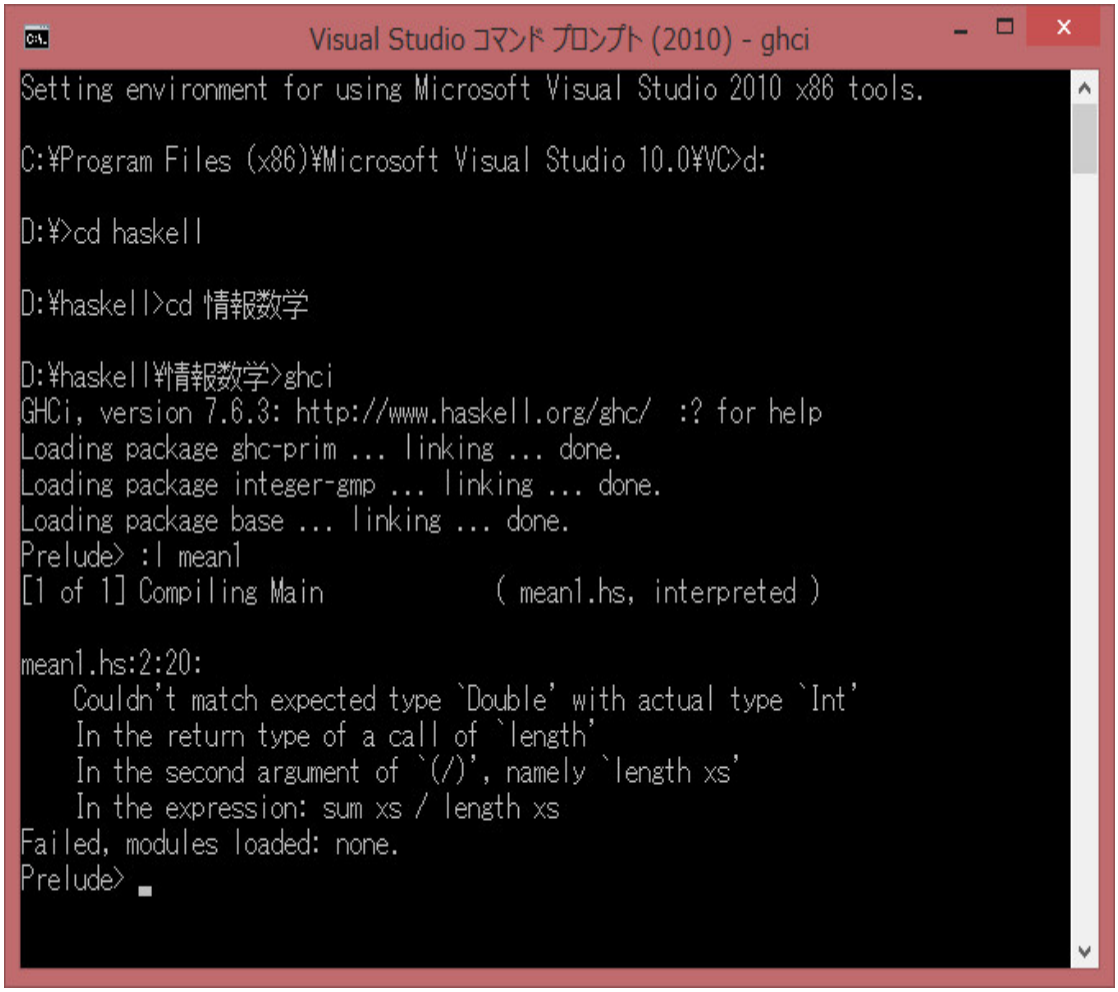
```
ghci
```

で、GHCI を起動した後

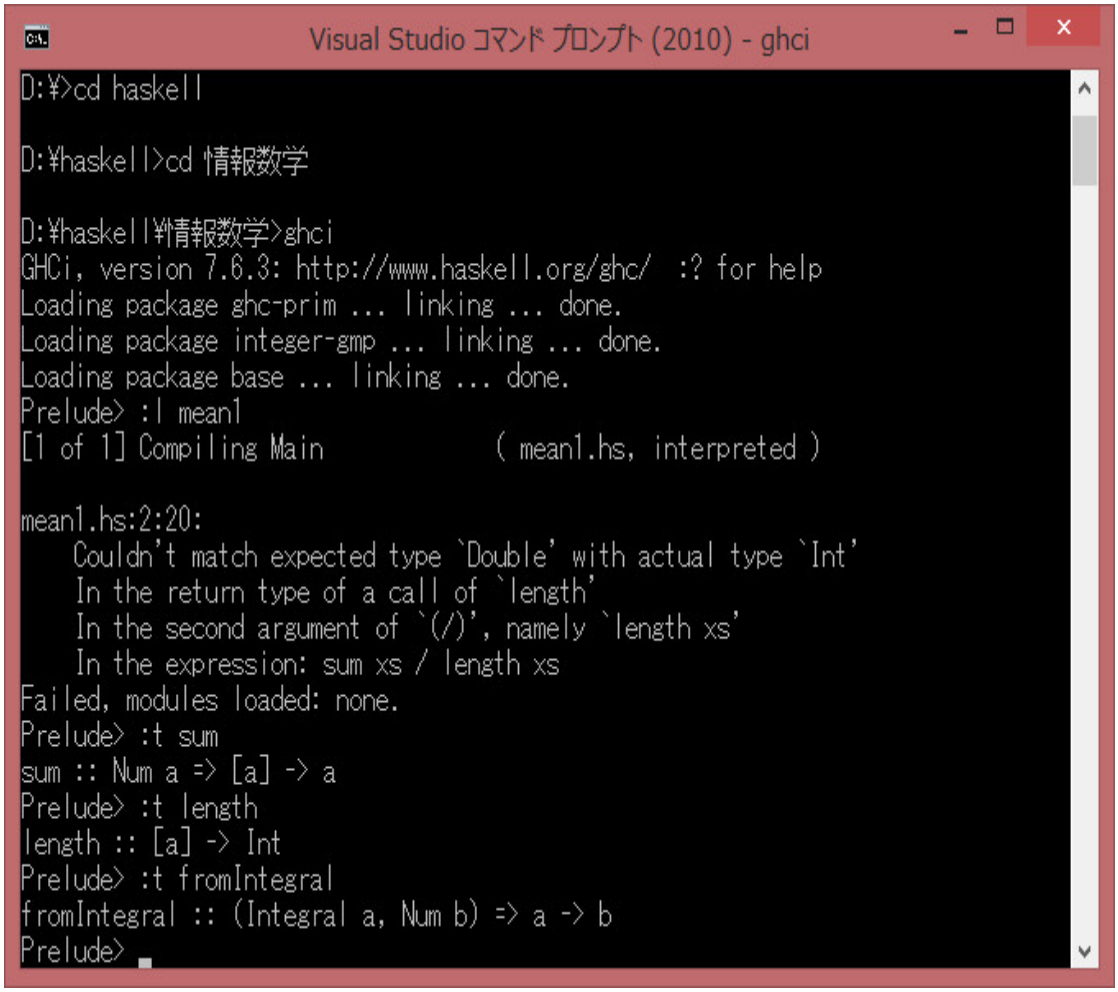
```
:l mean1
```

で、プログラムを読み込み、コンパイルします。

ところが、このプログラムを GHCI に読み込むとエラーになります。



です。sum length fromIntegral の定義を調べると次のようになります。



です。

Haskell の数の体系は一番上に数（Num）があり、演算子 (+,*,-) が定義されています。数（Num）の下に整数（Integral）と実数（Fractional）があり、整数（Integral）には演算子（div）が、実数（Fractional）には演算子（/）が定義されており、数（Num）で定義された演算子 (+,*,-) を引き継ぎます。整数（Integral）の下に固定長整数（Int）と多倍長整数（Integer）があり、演算子 (+,*,-,div) を引き継ぎます。実数（Fractional）の下に単精度浮動小数点数（Float）と倍精度浮動小数点数（Double）があり、演算子 (+,*,-,/) を引き継ぎます。数（Num）と整数（Integral）と実数（Fractional）は型クラスと言い、固定長整数（Int）と多倍長整数（Integer）と単精度浮動小数点数（Float）と倍精度浮動小数点数（Double）は型と言います。具体的なデータに対応するものが型で、複数の型に対して、共通の仕様（演算）を定義するものが型クラスです。型クラスの仕様は型クラスでも型でも引き継ぐことができますが、型の仕様は引き継ぐことができません。型 A が型クラス B の仕様を引き継いでいることを型 A は型クラス B のインスタンス（instance）であると言います。

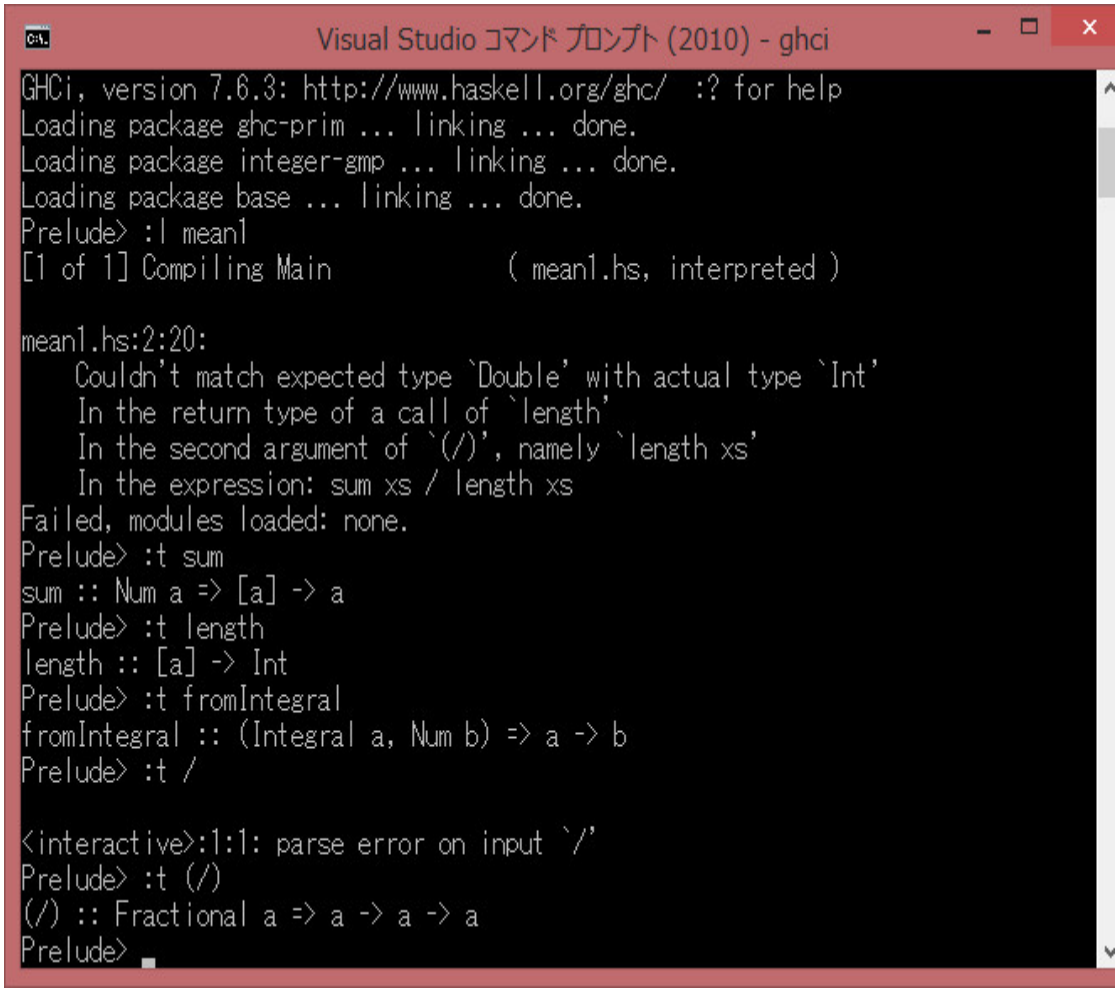
元に戻って、関数 sum は

```
sum :: Num a => [a] -> a
```

ですから、Num のインスタンスのリストから Num の値を計算する関数ですが、length は

```
length :: [a] -> Int
```

なので、任意の型のデータのリストから Int の値を計算する関数です。/ の定義を調べると



です。

```
(/) :: Fractional a => a -> a -> a
```

ですから、Fractional のインスタンスである値を二個引数に取って、Fractional のインスタンスである値を返す演算子ですから、Fractional のインスタンスで無い Int の値を引数にとれないと言って文句を言っていた訳です。この問題を解決するには、Int の値を Fractional のインスタンスとして認識できるように、Integral と Fractional の上位の Num のインスタンスとみなせるように、fromIntegral と言う関数で Int の値を Num の値に変換すればいいです。FromIntegral の定義は

```
fromIntegral :: (Integral a, Num b) => a -> b
```

です。

したがって、プログラムは

```
mean :: [Double] -> Double
mean xs = sum xs / fromIntegral   (length xs)
```

になります。サクラエディタを使って、上のプログラムを作り、mean1.hs という名前で適当なフォルダに保存し、コマンドプロンプトをそのフォルダに移動し、

```
ghci mean1
```

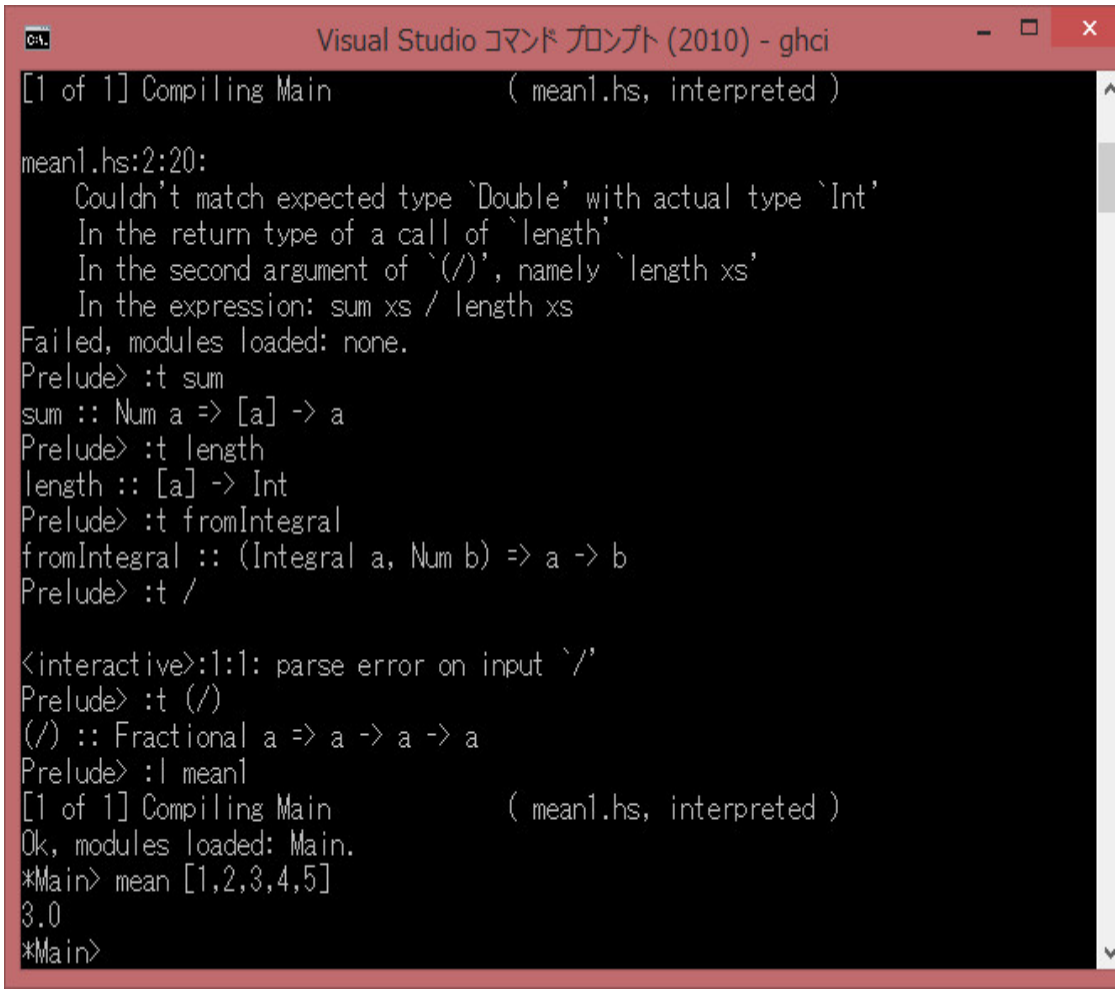
とするか

```
ghci
```

で、GHCI を起動した後

```
:l mean1
```

で、プログラムを読み込み、コンパイルします。文法的エラーがなければ、プロンプトが `Prelude>` から `Main>` になります。$mean\ [1,2,3,4,5]$ を実行すると

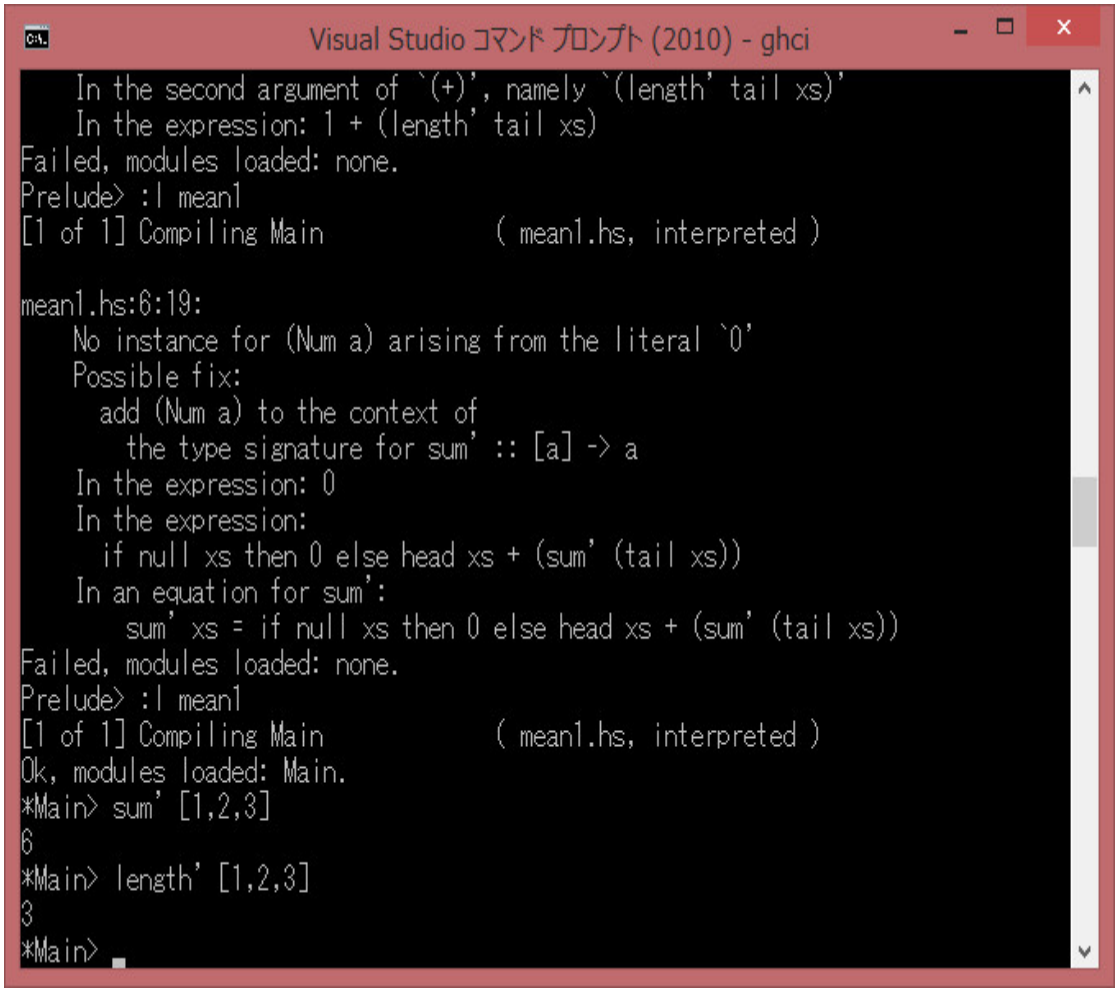


です。$[1,2,3,4,5]$ の平均値は 3.0 です。正しいみたいです。以下、プログラムの作り方や実行方法は同じなので繰り返しません。また、講義時間が限られていますので、以下のプログラムの細かな解説はしません。気になるなら、講義の時間に質問するか、自分で最後に挙げてある参考書を読んで勉強してください。関数型プログラミング言語のプログラミングがどんなものか雰囲気を解説するだけの文章です。

sum length は自分で作ることもできます。色々な方法があります。

```
sum' :: Num a => [a] -> a
sum' xs =
  if null xs then 0
  else head xs + (sum' (tail xs))

length' :: [a] -> Int
length' xs =
  if null xs then 0
  else 1 + (length' (tail xs))
```

sum と length はすでに定義されているので、sum' と length' としました。`if then else` を使って再帰的に定義しています。実行すると



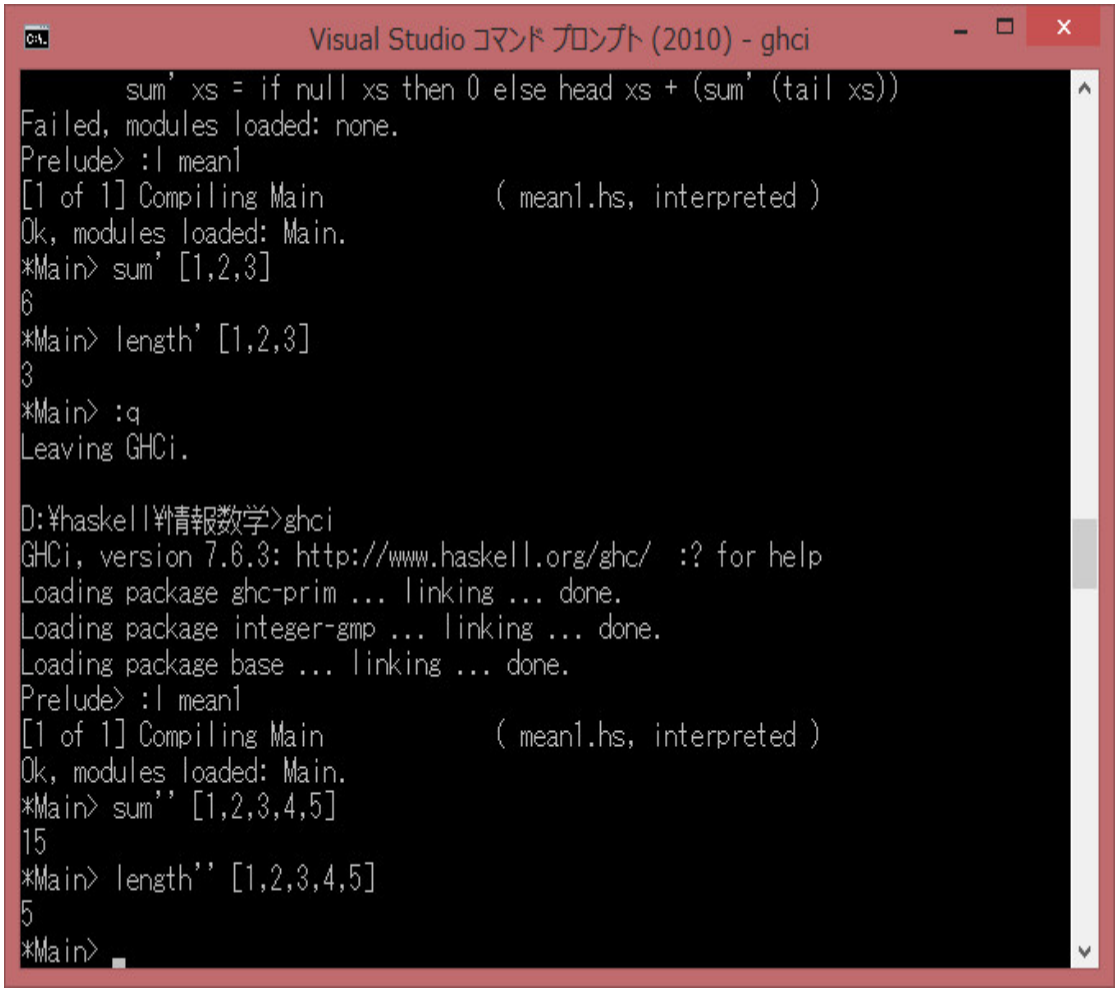
です。

パターン・マッチングを使って

```
sum'' :: Num a => [a] -> a
```

```
sum'' []     = 0
sum'' (x:xs) = x + (sum'' xs)

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

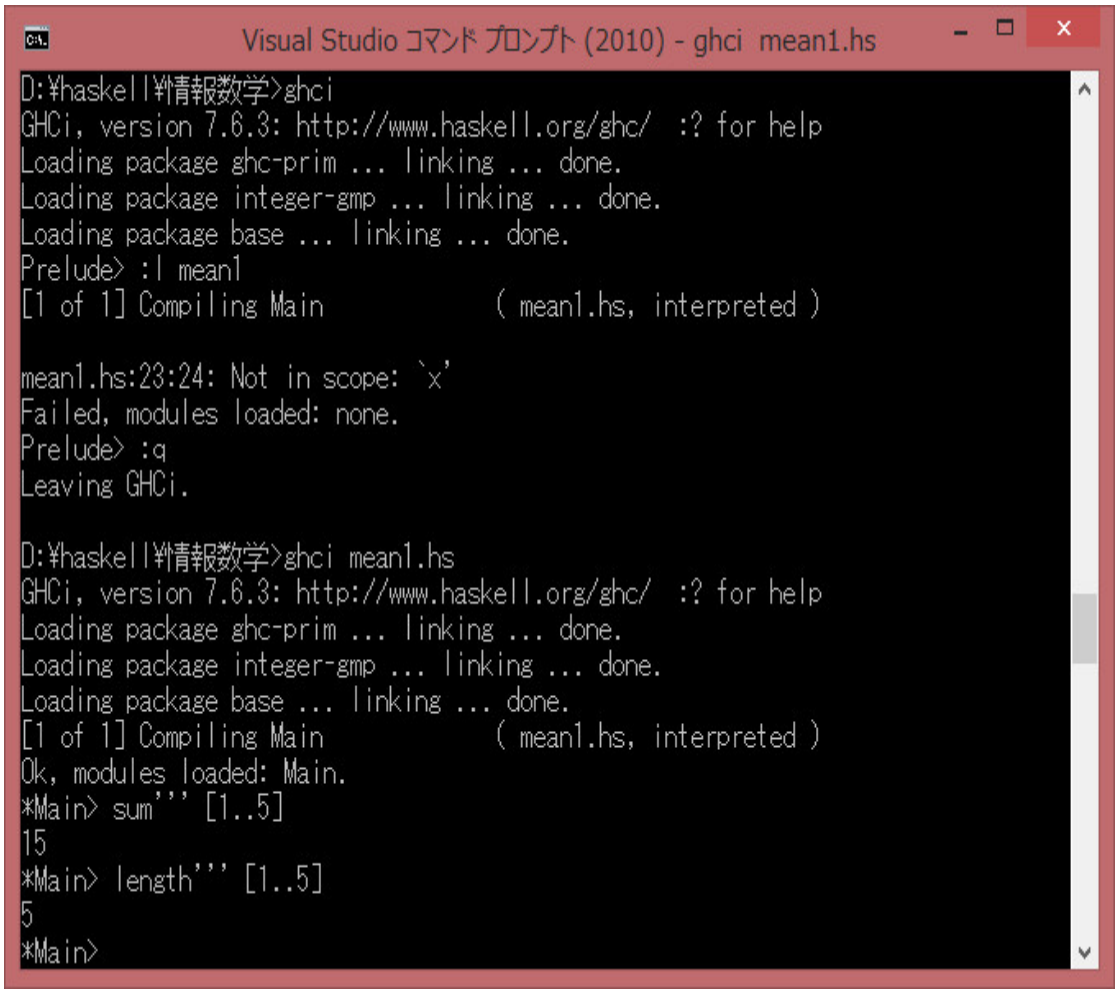
とすることもできます。実行すると



です。

```
sum''' :: Num a => [a] -> a
sum''' xs
  | null xs  = 0
  | otherwise =  (head xs) + (sum''' (tail xs))
length''' :: [a] -> Int
length''' xs
  | null xs   = 0
  | otherwise =  1 + (length''' (tail xs))
```

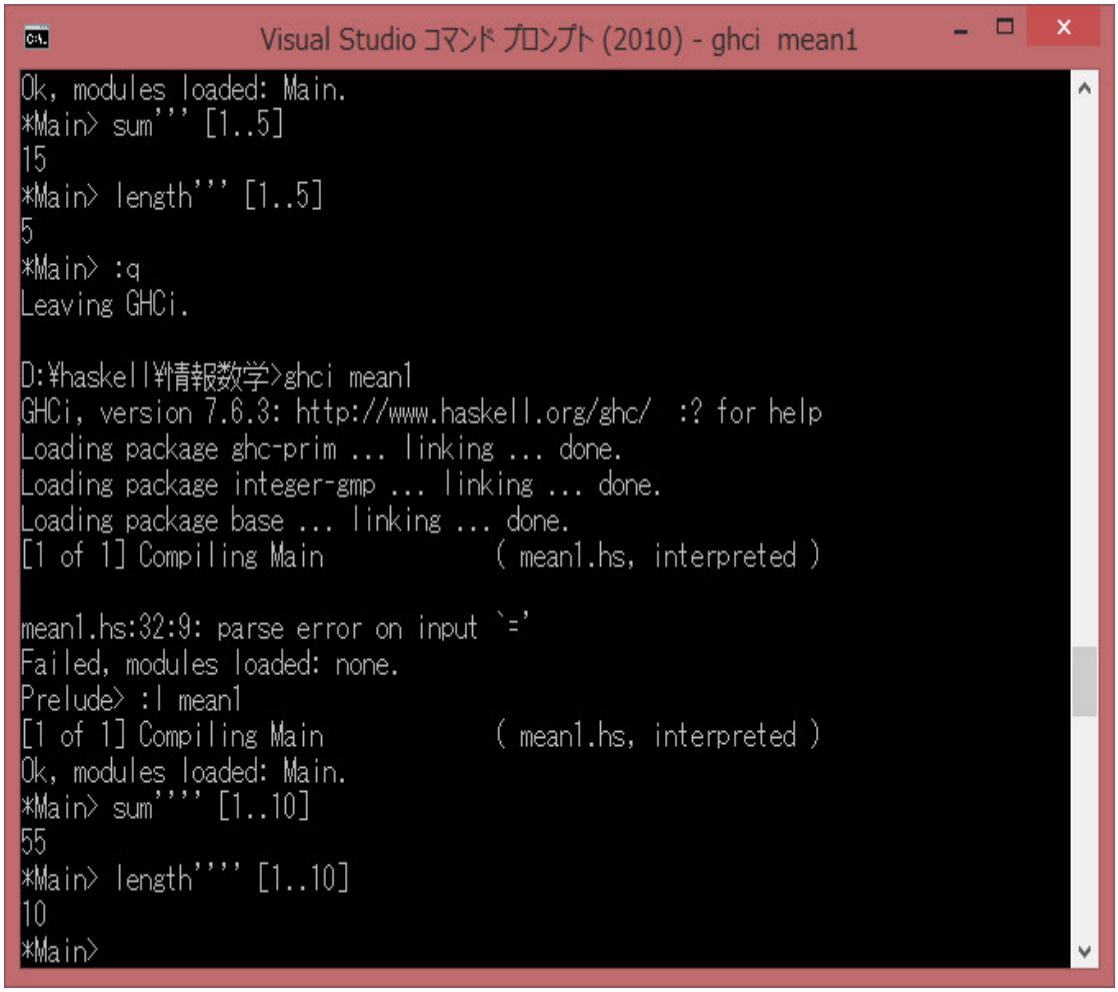
とすることもできます。実行すると



です。

```
sum'''' :: Num a => [a] -> a
sum'''' xs =
  case xs of
    []     -> 0
    (y:ys) ->  y + (sum'''' ys)
length'''' :: [a] -> Int
length'''' xs =
  case xs of
    []     -> 0
    (_:ys) ->  1 + (length'''' ys)
```

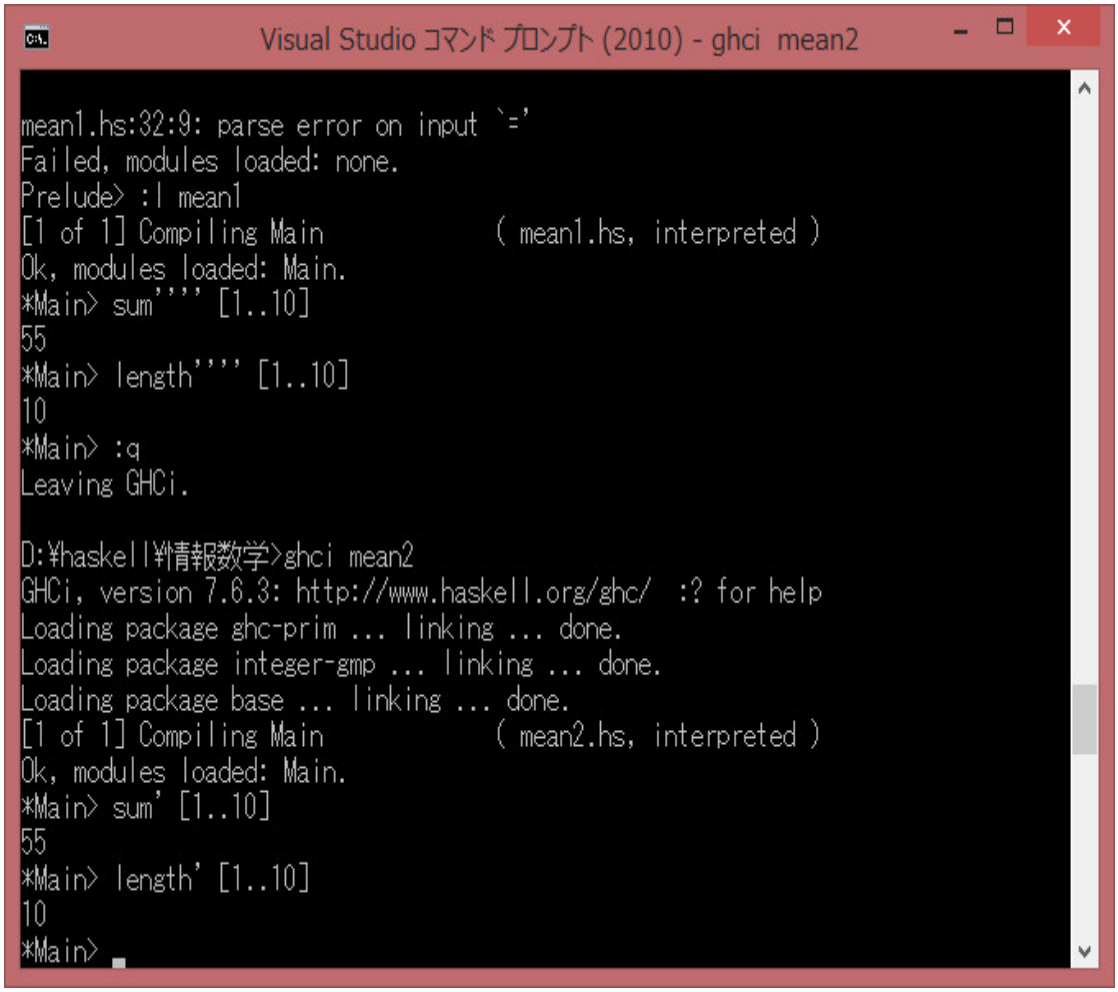
とすることもできます。実行すると



です。

```
sum' :: Num a => [a] -> a
sum' xs = iter xs 0
  where
    iter [] a = a
    iter (x:xs) a = iter xs (x + a)
length' :: [a] -> Int
length' xs = iter xs 0
  where
    iter [] a = a
    iter (x:xs) a = iter xs (a + 1)
```

とすることもできます。実行すると

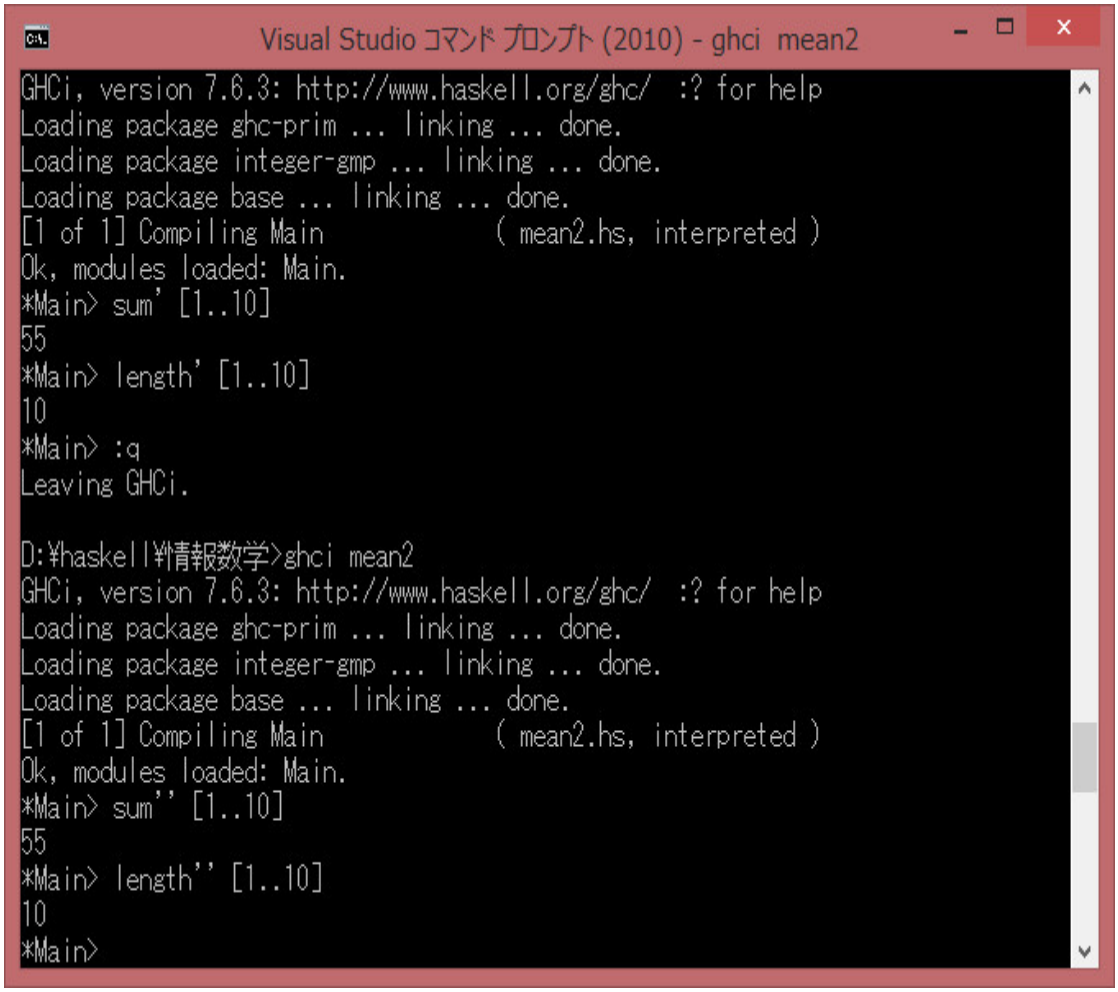


です。

畳み込みを使って

```
sum'' :: Num a => [a] -> a
sum'' xs = foldl (+) 0 xs

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

とすることもできます。実行すると

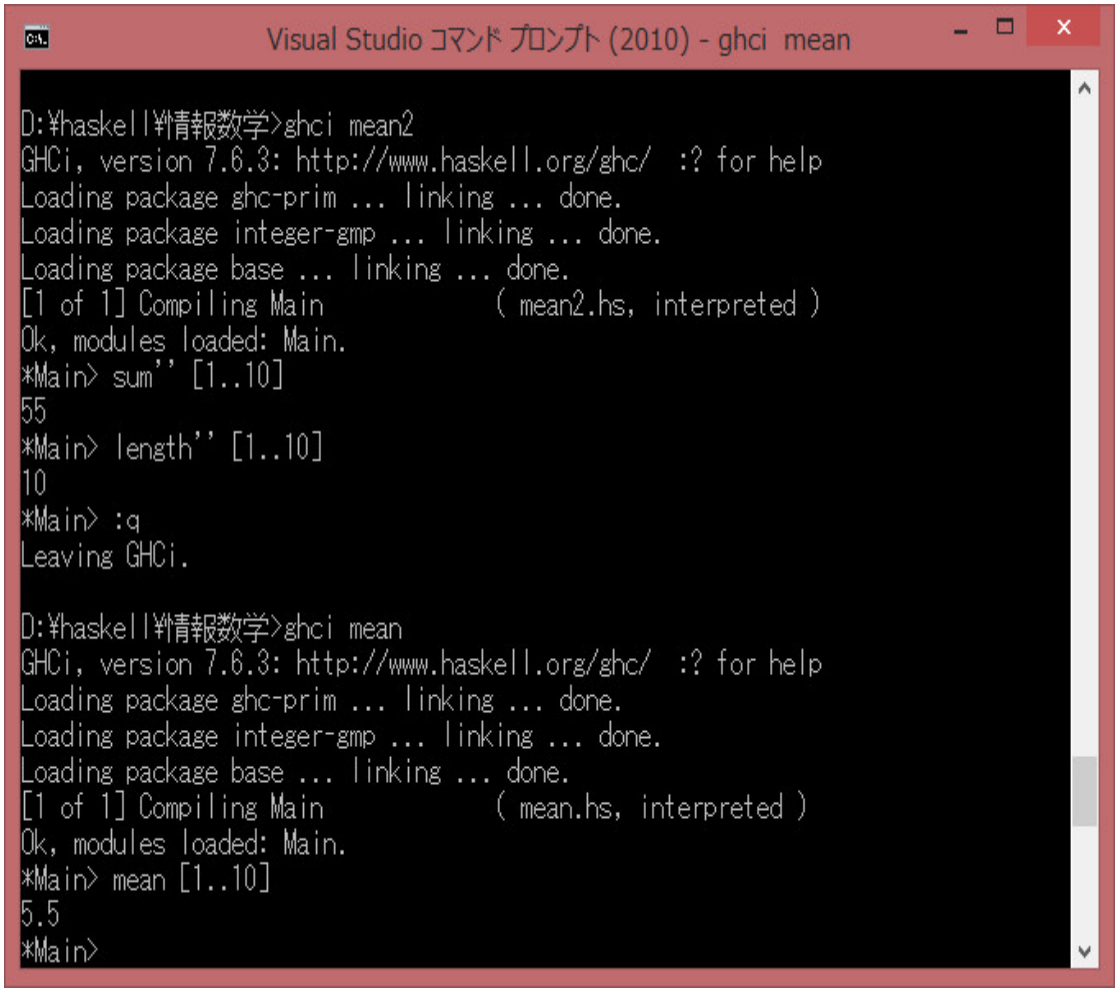


です。

従って、平均値を計算するプログラムは

```
mean :: [Double] -> Double
mean xs =
  let s = foldl (+) 0 xs
      l = foldl (\a _ -> a+1) 0 xs
  in s / l
```

とすることもできます。実行すると



です。

次に分散と標準偏差を求める関数を作ります。$\{x_1, x_2, \cdots, x_n\}$ の分散と標準偏差は

$$\mu = \frac{\sum_{k=1}^{n} x_k}{n}$$

$$\sigma^2 = \frac{\sum_{k=1}^{n} (x_k - \mu)^2}{n}$$

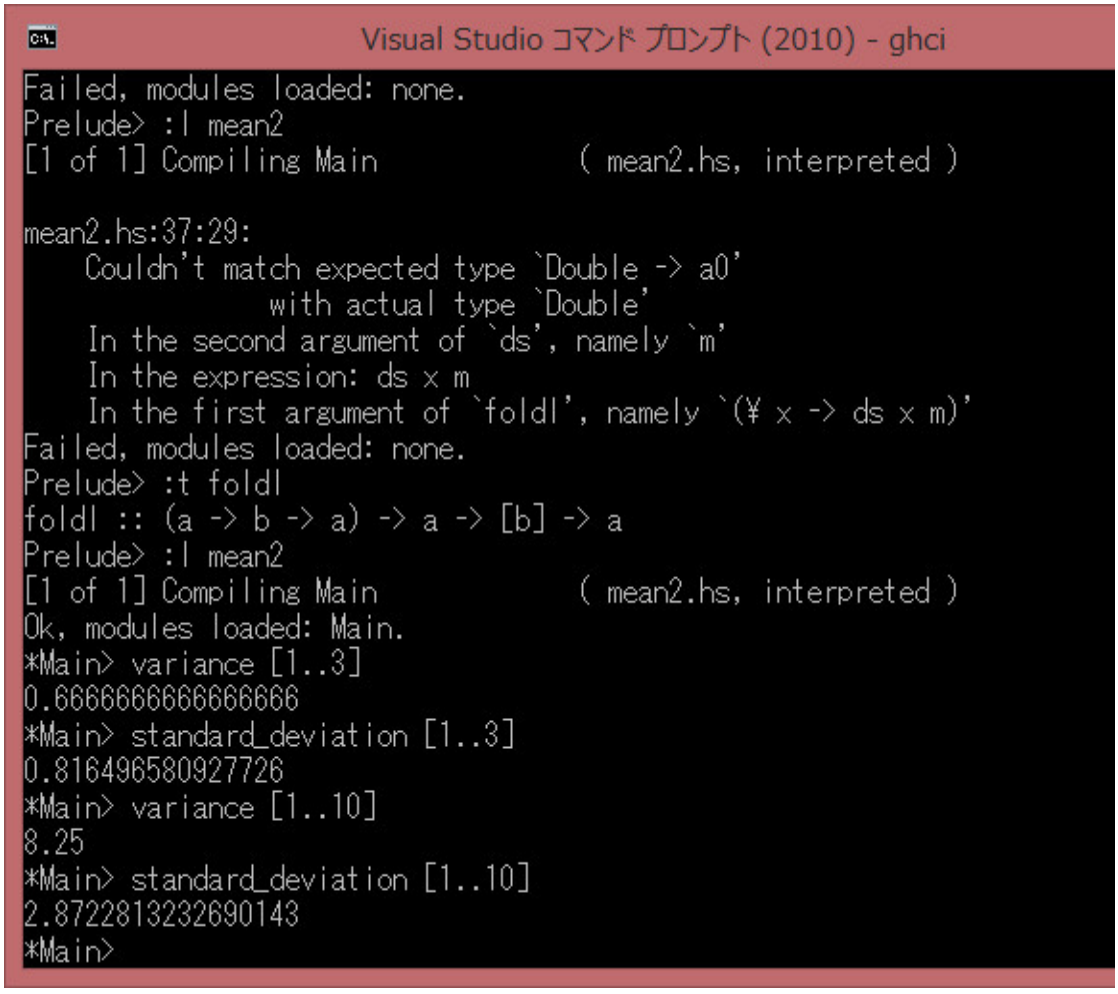
$$\sigma = \sqrt{\frac{\sum_{k=1}^{n} (x_k - \mu)^2}{n}}$$

で求められます。従って、分散と標準偏差を計算するプログラムは

```
mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)

variance :: [Double] -> Double
variance xs =
  let
    mu = mean xs
    ys = map (\x -> x - mu) xs
```

```
    zs = map (**2) ys
  in sum zs /  fromIntegral (length xs)

standard_deviation :: [Double] -> Double
standard_deviation xs = sqrt $ variance xs
```

とすることもできます。実行すると



です。畳み込みを使って

```
mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)

variance :: [Double] -> Double
variance xs =
  let m = mean xs
      n = fromIntegral (length xs)
      v = foldl (\a x -> a + (x-m)**2) 0 xs
```

```
  in v / n

standard_deviation :: [Double] -> Double
standard_deviation xs = sqrt $ variance xs
```

や

```
mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)

variance :: [Double] -> Double
variance xs = v / n where
  m = mean xs
  n = fromIntegral (length xs)
  v = foldl (\a x -> a + (x-m)**2) 0 xs

standard_deviation :: [Double] -> Double
standard_deviation xs = sqrt $ variance xs
```

とすることもできます。

高等学校では相関係数も学びます。相関係数を計算してみましょう。相関係数は

$$\text{相関係数} \quad = \frac{\text{共分散}}{x\text{の標準偏差}\cdot y\text{の標準偏差}}$$

です。共分散は

$$\text{共分散} = \frac{1}{n}\{(x_1-\overline{x})(y_1-\overline{y})+\cdots+(x_n-\overline{x})(y_n-\overline{y})\}$$

です。したがって、平均値、標準偏差を計算する関数は作っているので、共分散を計算する関数を作る必要があります。

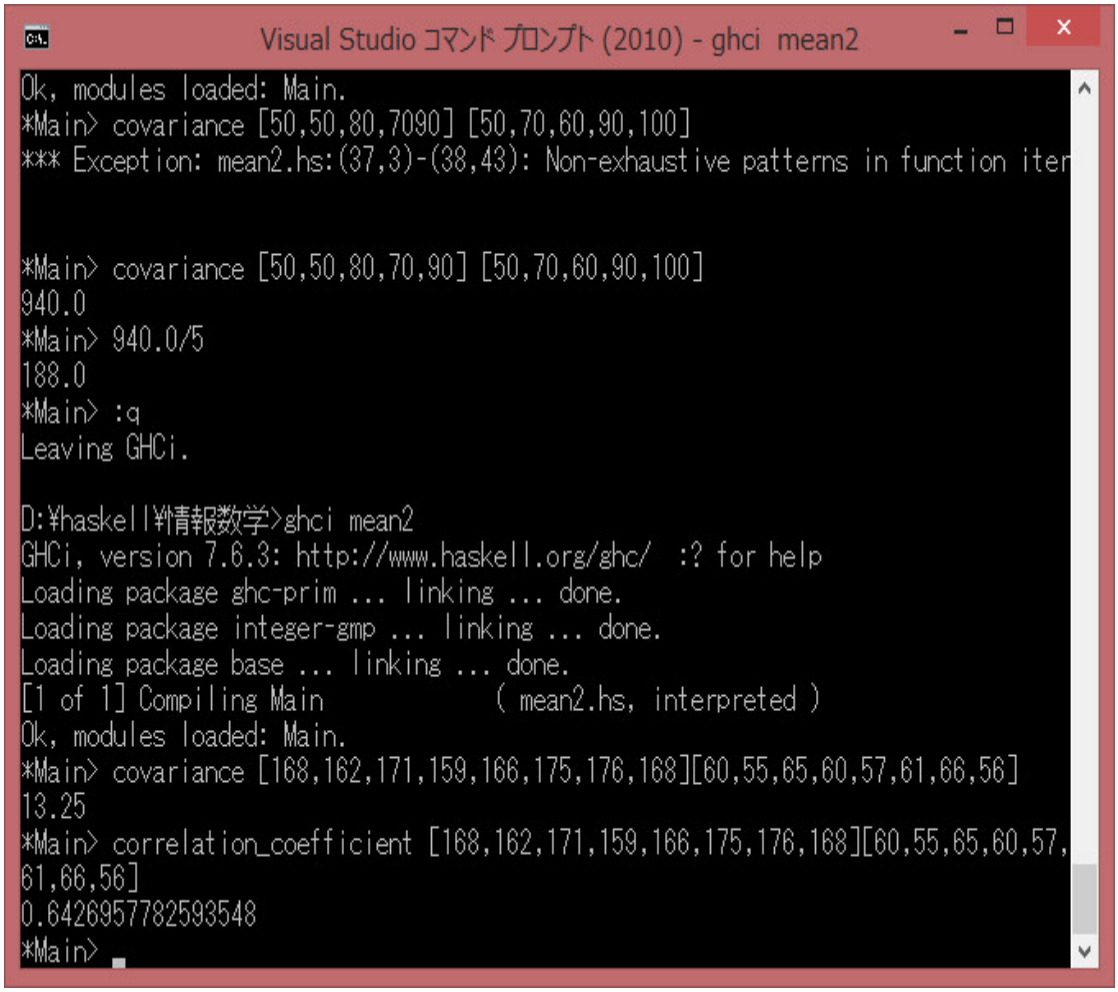
```
mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)

variance :: [Double] -> Double
variance xs =
  let
    mu = mean xs
    ys = map (\x -> x - mu) xs
    zs = map (**2) ys
  in sum zs /  fromIntegral (length xs)
standard_deviation :: [Double] -> Double
standard_deviation xs = sqrt $ variance xs

covariance :: [Double] -> [Double] -> Double
covariance xs ys = iter xs2 ys2 0 where
  mx = mean xs
```

```
  my = mean ys
  xs2 = map (\x -> x - mx) xs
  ys2 = map (\y -> y - my) ys
  iter [] [] a = a / fromIntegral (length xs)
  iter (u:us) (v:vs) a = iter us vs (u*v+a)
```

従って、相関係数を計算する関数は

```
mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)

variance :: [Double] -> Double
variance xs =
  let
    mu = mean xs
    ys = map (\x -> x - mu) xs
    zs = map (**2) ys
  in sum zs /  fromIntegral (length xs)
standard_deviation :: [Double] -> Double
standard_deviation xs = sqrt $ variance xs

covariance :: [Double] -> [Double] -> Double
covariance xs ys = iter xs2 ys2 0 where
  mx = mean xs
  my = mean ys
  xs2 = map (\x -> x - mx) xs
  ys2 = map (\y -> y - my) ys
  iter [] [] a = a / fromIntegral (length xs)
  iter (u:us) (v:vs) a = iter us vs (u*v+a)

correlation_coefficient ::  [Double] -> [Double] -> Double
correlation_coefficient xs ys =
  let cov = covariance xs ys
      sdx = standard_deviation xs
      sdy = standard_deviation ys
  in (cov / sdx) / sdy
```

でプログラミングできます。実行すると



です。さらに、関数 zipWith を使って

```
mean :: [Double] -> Double
mean xs = sum xs / fromIntegral (length xs)

variance :: [Double] -> Double
variance xs =
  let
    mu = mean xs
    ys = map (\x -> x - mu) xs
    zs = map (**2) ys
  in sum zs /  fromIntegral (length xs)
standard_deviation :: [Double] -> Double
standard_deviation xs = sqrt $ variance xs

covariance :: [Double] -> [Double] -> Double
covariance xs ys = cov / n where
  mx = mean xs
```

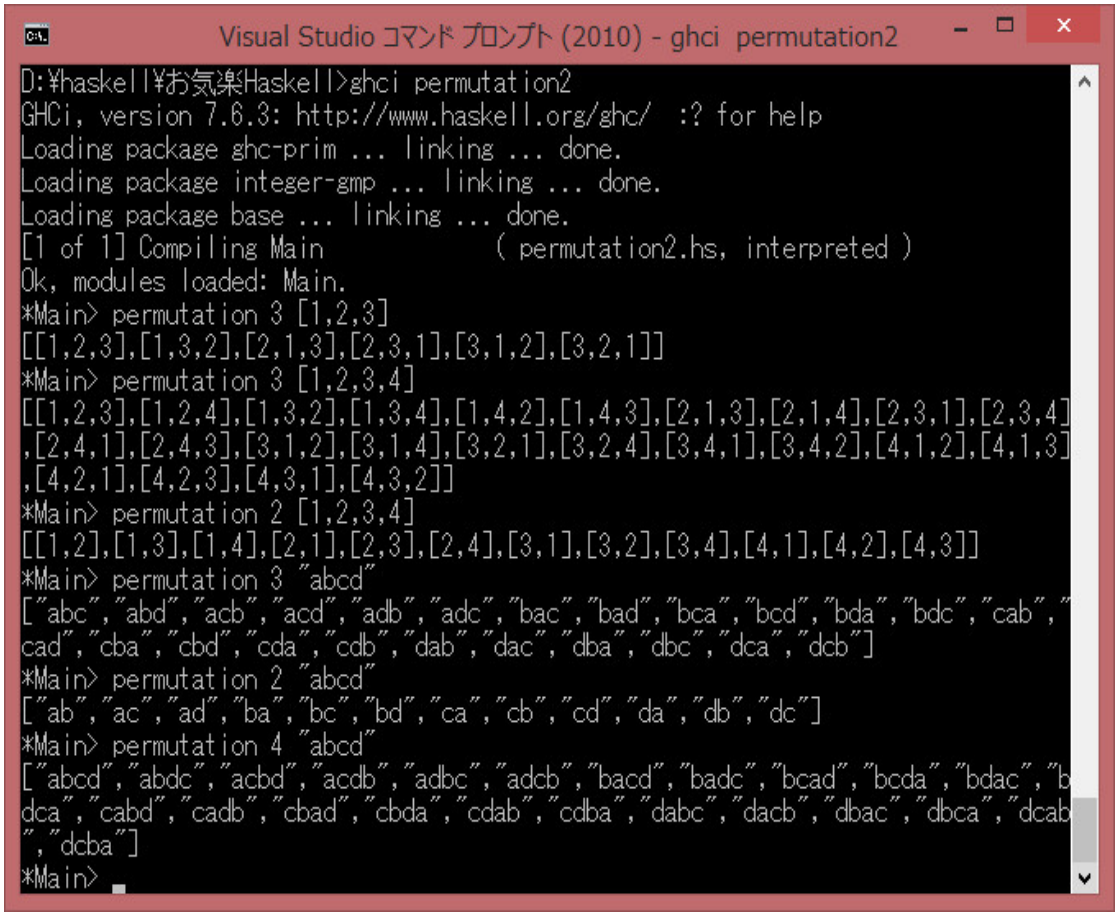
```
  my = mean ys
  xs2 = map (\x -> x - mx) xs
  ys2 = map (\y -> y - my) ys
  cov = sum $ zipWith (*) xs2 ys2
  n = fromIntegral (length xs)

correlation_coefficient ::  [Double] -> [Double] -> Double
correlation_coefficient xs ys =
  let cov = covariance xs ys
      sdx = standard_deviation xs
      sdy = standard_deviation ys
  in (cov / sdx) / sdy
```

とプログラミングすることもできます。

順列と組み合わせを高等学校で学びます。これを求めるプログラムを作ります。まず、順列を求めるプログラムは色々考えられますが、一つの例は次のようになります。

```
import Control.Monad

permutation :: Eq a => Int -> [a] -> [[a]]
permutation n xs = iter 0 [] [] where
  iter m ys zs
    | m == n    = reverse ys : zs
    | otherwise = do
      x <- xs
      guard (x `notElem` ys)
      iter (m+1) (x:ys) zs
```

実行すると

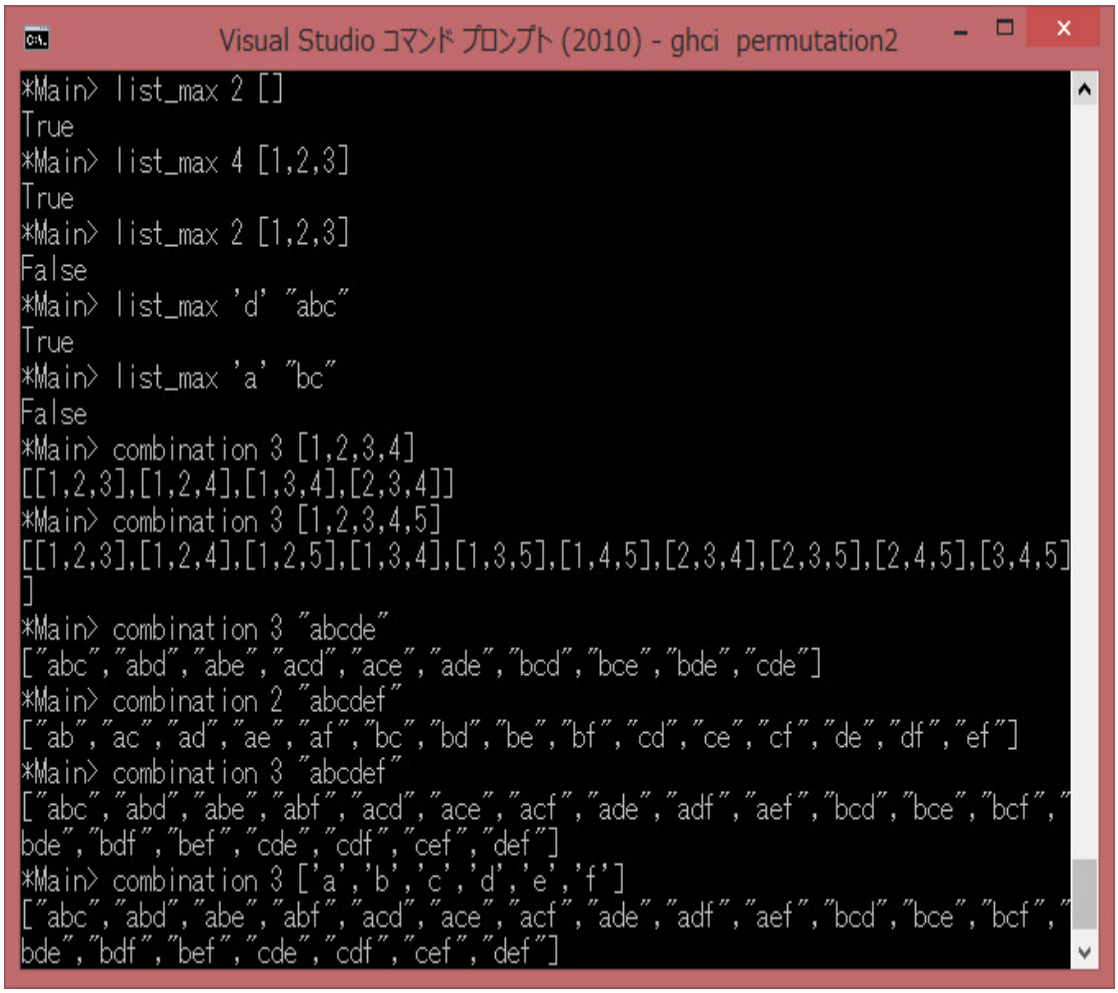


です。

組み合わせを求めるプログラムも色々考えられますが、一つの例は次のようになります。

```
import Control.Monad

list_max :: Ord a => a -> [a] -> Bool
list_max _ [] = True
list_max x (y:ys) = if x <= y then False else list_max x ys

combination :: Ord a => Int -> [a] -> [[a]]
combination n xs = iter 0 [] [] where
  iter m ys zs
    | m == n    = reverse ys : zs
    | otherwise = do
      x <- xs
      guard (list_max x ys)
      iter (m+1) (x:ys) zs
```

実行すると



です。

　中国語の発音練習の本を読んでいると発音の練習があって、

第一声　第二声　第三声　第四声
第二声　第三声　第四声　第一声
第三声　第四声　第一声　第二声
第四声　第一声　第二声　第三声

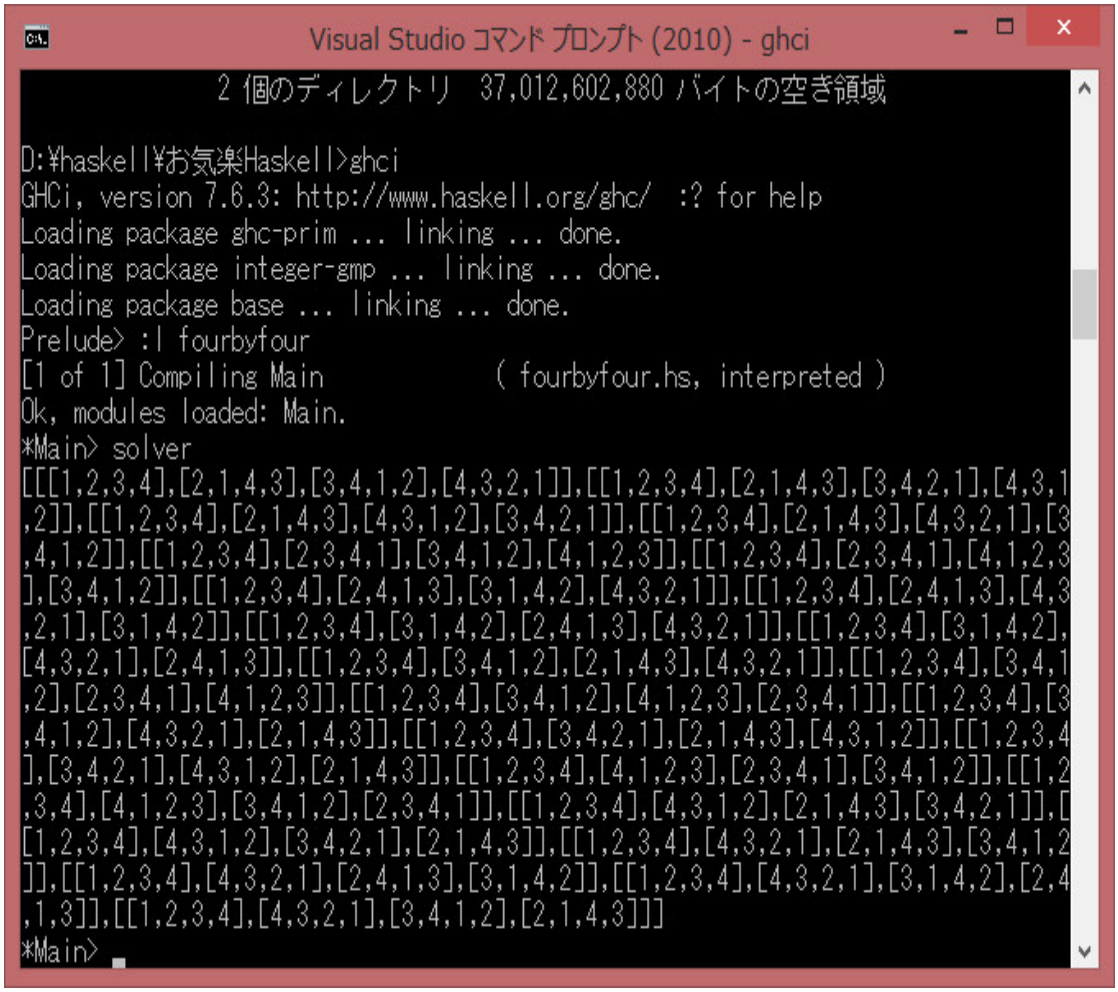
の順に発音を練習するようになっていました。このように縦横に第一声、　第二声、　第三声、第四声が一つずつ並ぶ並べ方をすべて求めてみましょう。

　$[1,2,3,4]$ のすべての順列の集合を求め、最初は $[1,2,3,4]$ とし、そのの順列の集合から後３個を選び、条件を満たす４組の集合を作ればいいです。プログラムは次のようになります。

```
import Control.Monad

permutation :: Eq a => Int -> [a] -> [[a]]
permutation n xs = iter 0 [] where
  iter m ys
```

```
    | m == n    = return (reverse ys)
    | otherwise = do
      x <- xs
      guard (x `notElem` ys)
      iter (m+1) (x:ys)

solver :: [[[Int]]]
solver  = iter (permutation 4 [1,2,3,4]) [] where
  iter source xs = do
    x1 <- [[1,2,3,4]]
    x2 <- source
    x3 <- source
    x4 <- source
    y1 <- [[x1 !! 0, x2 !! 0, x3 !! 0, x4 !! 0]]
    guard(y1 `elem` source)
    y2 <- [[x1 !! 1, x2 !! 1, x3 !! 1, x4 !! 1]]
    guard(y2 `elem` source)
    y3 <- [[x1 !! 2, x2 !! 2, x3 !! 2, x4 !! 2]]
    guard(y3 `elem` source)
    y4 <- [[x1 !! 3, x2 !! 3, x3 !! 3, x4 !! 3]]
    guard(y4 `elem` source)
    [x1, x2, x3, x4] : xs
```

実行すると



です。

順列を求めるプログラムを作ったのでそれを使って、行列式を定義通り計算してみましょう。必要な関数を順番に作っていきます。順列を計算する関数は作っているので、偶置換か奇置換かを判定する関数を作ります。

順列 $J = \{j_1, j_2, j_3, \cdots, j_n\}$ において、$m = 1.2, \cdots, n-1$ に対して項 j_m の後ろにある項 $j_l, m+1 \leq l \leq n$ のうちで j_m より値が小さい項の個数 $p(j_m)$ の総和

$$p(J) = p(j_1) + p(j_2) + p(j_3) + \cdots + p(j_{n-1})$$

が偶数であれば偶置換で、奇数であれば奇置換なのでこれを使うことにします。

```
g :: Int -> [Int] -> Int
g _ [] = 0
g x (y:ys) = if x > y then 1 + g x ys else g x ys

p :: [Int] -> Int
p [] = 0
p (x:[]) = 0
p (x:xs) = (g x xs) + p xs
```

```
sign :: [Int] -> Int
sign xs = if even (p  xs) then 1 else -1
```

のプログラムで、$g\ j_m\ [j_{m+1}, j_{m+2}, \cdots, j_n]$ は項 j_m の後ろにある項 $j_l, m+1 \leq l \leq n$ のうちで j_m より値が小さい項の個数を計算し、この g を補助関数として、$p\ [j_1, j_2, j_3, \cdots, j_n]$ が順列 $J = \{j_1, j_2, j_3, \cdots, j_n\}$ において、$m = 1.2, \cdots, n-1$ に対して項 j_m の後ろにある項 $j_l, m+1 \leq l \leq n$ のうちで j_m より値が小さい項の個数 $p(j_m)$ の総和

$$p(J) = p(j_1) + p(j_2) + p(j_3) + \cdots + p(j_{n-1})$$

を計算し、$sign\ [j_1, j_2, j_3, \cdots, j_n]$ が $[j_1, j_2, j_3, \cdots, j_n]$ が 偶置換であれば 1 、奇数であれば－ 1 を返す関数です。

n 次正方行列 A の行列式は A の (i,j) 成分を a_{ij} とすると

$$det(A) = \sum_{J \in P_n} sign(J) a_{1j_1} a_{2j_2} a_{3j_3} \cdots a_{nj_n}$$

です。つまり。$[1, 2, 3, \cdots, n]$ のすべての順列の集合 P_n を作り、各順列 $J = [j_1, j_2, j_3, \cdots, j_n]$ に対して

$$sign([j_1, j_2, j_3, \cdots, j_n]) a_{1j_1} a_{2j_2} a_{3j_3} \cdots a_{nj_n}$$

を計算し、それらの和が行列式であるが我々が普通習う行列式の定義です。

行列式を計算するために必要な残りの関数は次のようになります。

```
h' :: Int -> Int -> [Int] -> [[Double]] -> Double
h' m n js xs
  | m == n    = 1
  | otherwise = ((xs !! m) !! (head js -1)) * h' (m+1) n (tail js) xs

det' :: Int -> [[Int]] -> [[Double]] -> Double
det' _ [] _ = 0
det' n (z:zs) xs = fromIntegral (sign z) * h' 0 n z xs + det' n zs xs

det xs = det' n perm xs
  where
    n = length xs
    perm = permutation n [1 .. n]
```

行列を $[[1, 2], [2, 3]]$ や $[[1, 2, 3], [4, 5, 6], , [7, 8, 9]]$ のようなリストで表し、$det\ [[1, 2, 3], [4, 5, 6], , [7, 8, 9]]$ で 、行列 $[[1, 2, 3], [4, 5, 6], , [7, 8, 9]]$ の行列式を計算します。このために行列の次数 n と $[1, 2, \cdots, n]$ の順列の集合を計算し、$det\ [[1, 2, 3], [4, 5, 6], , [7, 8, 9]]$ の場合、補助関数 det' を使って、

$det'\ 3\ [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]\ [[1, 2, 3], [4, 5, 6], , [7, 8, 9]]$

で計算します。そして、補助関数 h' を使って

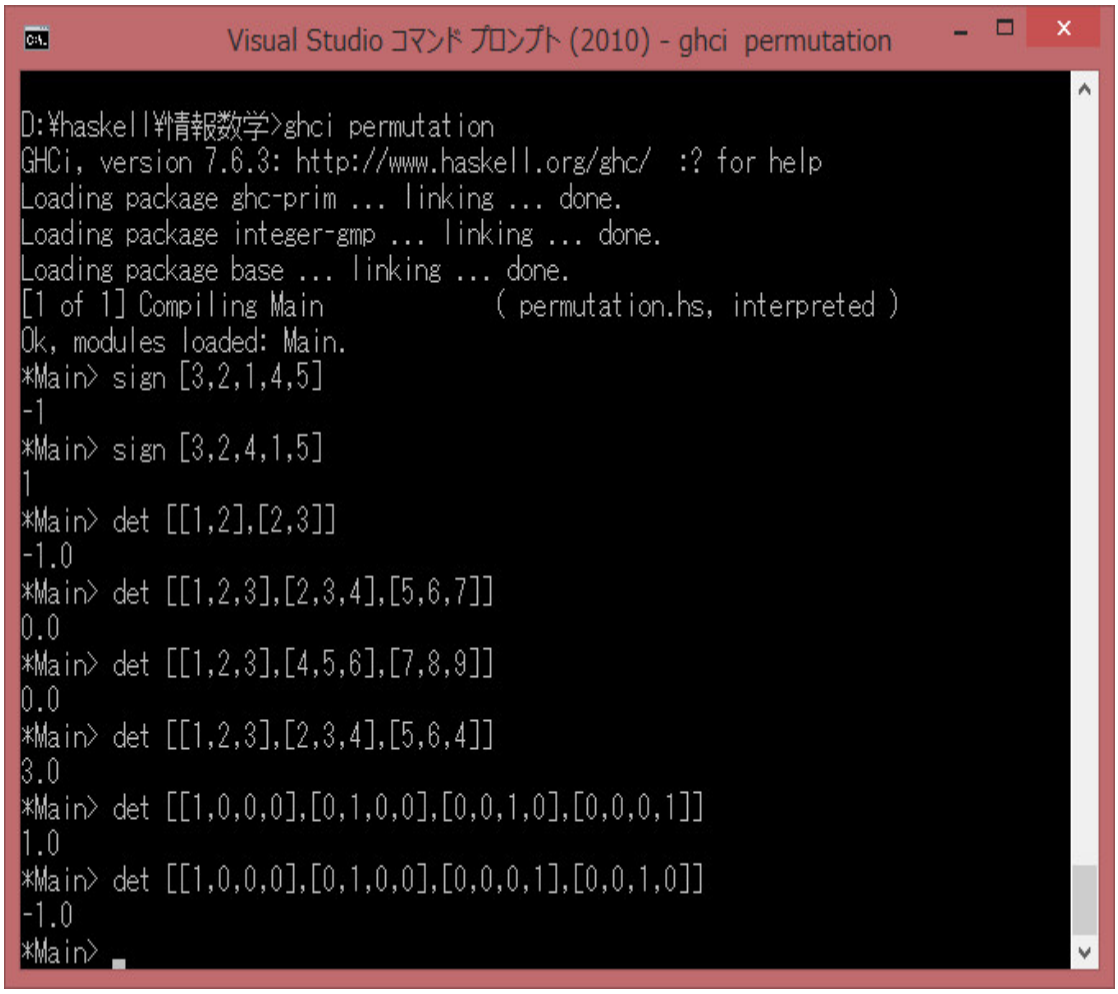
$$sign([j_1, j_2, j_3, \cdots, j_n]) a_{1j_1} a_{2j_2} a_{3j_3} \cdots a_{nj_n}$$

を計算します。プログラムの全体は

```
import Control.Monad

permutation :: Eq a => Int -> [a] -> [[a]]
permutation n xs = iter 0 [] [] where
  iter m ys zs
    | m == n    = reverse ys : zs
    | otherwise = do
      x <- xs
      guard (x `notElem` ys)
      iter (m+1) (x:ys) zs

list_max :: Ord a => a -> [a] -> Bool
list_max _ [] = True
list_max x (y:ys) = if x <= y then False else list_max x ys

combination :: Ord a => Int -> [a] -> [[a]]
combination n xs = iter 0 [] [] where
  iter m ys zs
    | m == n    = reverse ys : zs
    | otherwise = do
      x <- xs
      guard (list_max x ys)
      iter (m+1) (x:ys) zs

g :: Int -> [Int] -> Int
g _ [] = 0
g x (y:ys) = if x > y then 1 + g x ys else g x ys

p :: [Int] -> Int
p [] = 0
p (x:[]) = 0
p (x:xs) = (g x xs) + p xs

sign :: [Int] -> Int
sign xs = if even (p  xs) then 1 else -1

h' :: Int -> Int -> [Int] -> [[Double]] -> Double
h' m n js xs
  | m == n    = 1
  | otherwise = ((xs !! m) !! (head js -1)) * h' (m+1) n (tail js) xs

det' :: Int -> [[Int]] -> [[Double]] -> Double
det' _ [] _ = 0
```

```
det' n (z:zs) xs = fromIntegral (sign z) * h' 0 n z xs + det' n zs xs

det xs = det' n perm xs
  where
    n = length xs
    perm = permutation n [1 .. n]
```

です。実行すると



です。

私は自分が必要なプログラムは基本的には手続き型プログラミング言語（最近はもっぱら C++）で作りますので、関数型プログラミングは得意ではなく、Lisp を勉強したときは、マイコンの性能が低く Lisp でプログラミングするのではなく Lisp インタプリタの作り方ばかり勉強していたし、Scheme や Haskell はまだまだプログラミングの経験が少ないので、 Haskell の参考書にあるような、自慢できるような簡潔なプログラムはよう作りませんが、このように必要な関数を次々作り、プログラムをでっちあげればいいです。どんなものでも正しく動くものを作り上げることができれば自信になります。参考書の例題により Haskell の基本的な文法と文化を理解したら、行列式のように計算するための公式やアルゴリズムを理解している関数のプログラムを片っ端から自分で

作ってみます。お手本のプログラムを読んで、打ち込んで実行するだけでは、失敗しないので真の理解にはなりません。例題のプログラムを暗記する事が勉強ではありません。この行列式を計算するプログラムのようなプログラムを自分で作ってみると色々な失敗をします。どうしてコンパイルできないか、実行時にエラーが起こって止まってしまうのか、エラーの情報を見ながら直すことが本当の勉強になります。Haskell はコンパイラもありますので、文法的に正しいプログラムが出来上がらないと実行できません。失敗すればいろいろなことを理解することができます。失敗を繰り返すことにより、 Haskell の基本的な文法と文化を真に理解することができます。沢山簡単なプログラムを作ってみます。大きなプログラムに必要な各々の関数を正しく動いていることをチェックしながら、作ります。そしてこれらを纏めて作りたかったプログラムを組み立てます。これをボトムアップの手法と言います。手続き型プログラミングではトップダウンの手法を使いますが、関数型プログラミングではこのようにボトムアップの手法を使います。

良く知られた行列式を計算するアルゴリズムは Python の解説に載せています。行列式が計算できたので、連立方程式をクラメール (Cramer) の公式を使って計算してみましょう。クラメール (Cramer) の公式というのは

$$\begin{cases} 2x + 4y = 10 & (1) \\ x + 3y = 6 & (2) \end{cases}$$

は、行列を使って

$$\begin{pmatrix} 2 & 4 \\ 1 & 3 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 10 \\ 6 \end{pmatrix}$$

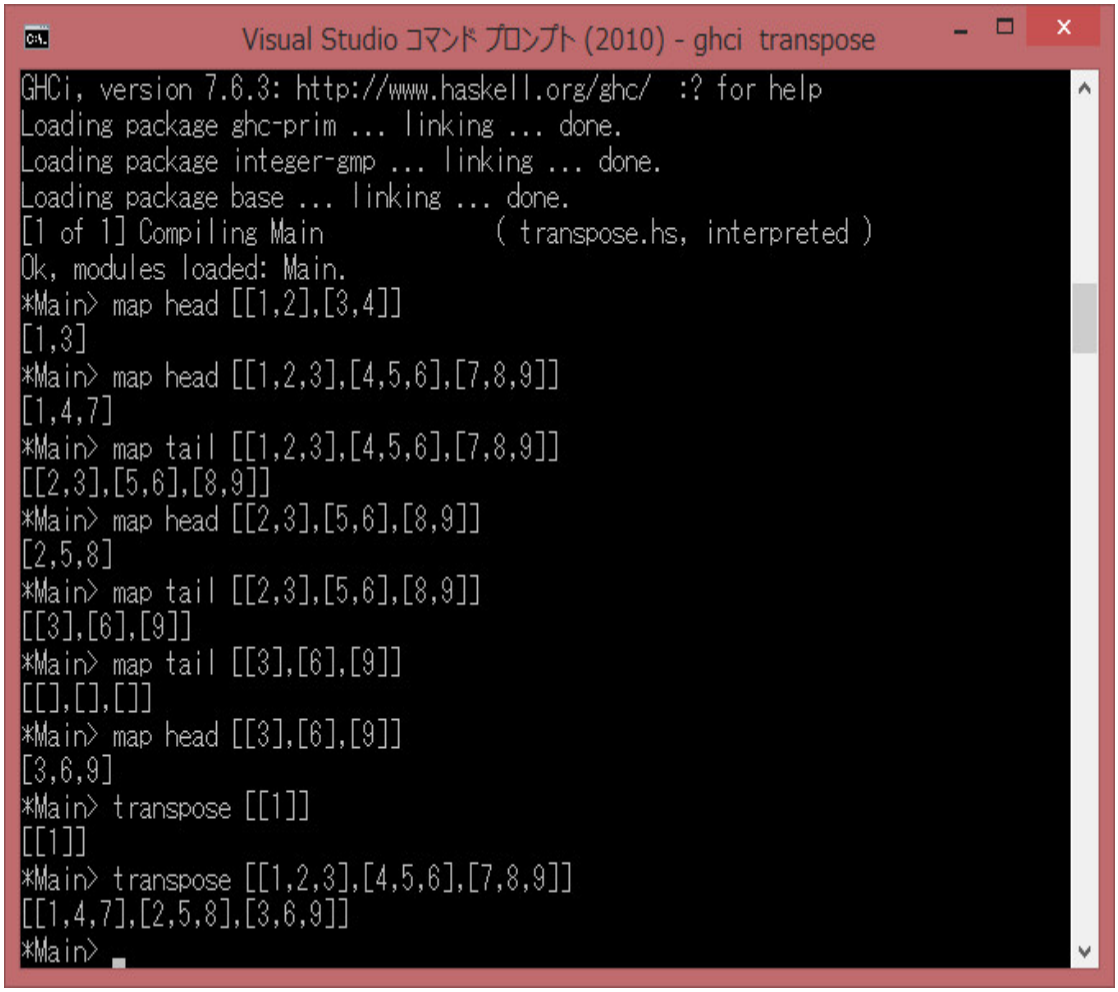
と表現して、

$$x = \frac{\begin{vmatrix} 10 & 4 \\ 6 & 3 \end{vmatrix}}{\begin{vmatrix} 2 & 4 \\ 1 & 3 \end{vmatrix}} \quad y = \frac{\begin{vmatrix} 2 & 10 \\ 1 & 6 \end{vmatrix}}{\begin{vmatrix} 2 & 4 \\ 1 & 3 \end{vmatrix}}$$

を計算しなさいと言うものです。行列式を計算する関数は作っているので、行列の要素を入れ替える関数を作れば完成です。列を入れ替える関数を強引に作ってもいいですが、転置行列を与える関数を作って、行を入れ替える方が楽な気がします。「車輪は再発見するな」という言葉があります。転置行列を与える関数を誰か作っていないかインターネットで調べてみます。在りました。

```
transpose :: [[a]] -> [[a]]
transpose ([]:_) = []
transpose xs = map head xs : transpose (map tail xs)
```

これで良いと言うのです。確かめてみます。



簡潔な良いプログラムです。どなたが作ったかお名前はわかりませんがこれを使わしていただくことにします。後は

$$change\ [a_1, a_2, \cdots, a_n]\ b = [[b, a_2, \cdots, a_n], [a_1, b, \cdots, a_n], \cdots, [a_1, a_2, \cdots, b]]$$

を作ります。このために補助関数

$$change'\ k\ [a_1, a_2, \cdots, a_k, , \cdots, a_n]\ b = [a_1, a_2, \cdots, b, , \cdots, a_n] \quad (0 \leq k \leq n-1)$$

を使えば良いです。

```
change' :: Int -> [a] -> a -> [a]
change' 0 (x:xs) y = y : xs
change' k (x:xs) y = x : change' (k-1) xs y
change :: [a] -> a -> [[a]]
change xs y = iter 0 n xs y where
  n = length xs
  iter k n xs y
    | k == n = []
    | otherwise = change' k xs y :iter (k+1) n xs y
```

で良いです。従って、最後に、連立方程式を解くプログラムは

```
solver :: [[Double]] -> [Double] -> [Double]
solver xs ys =
  let zs = transpose xs
      ws = change zs ys
      us = map transpose ws
      d  = det xs
  in map (\a -> (det a) / d) us
```

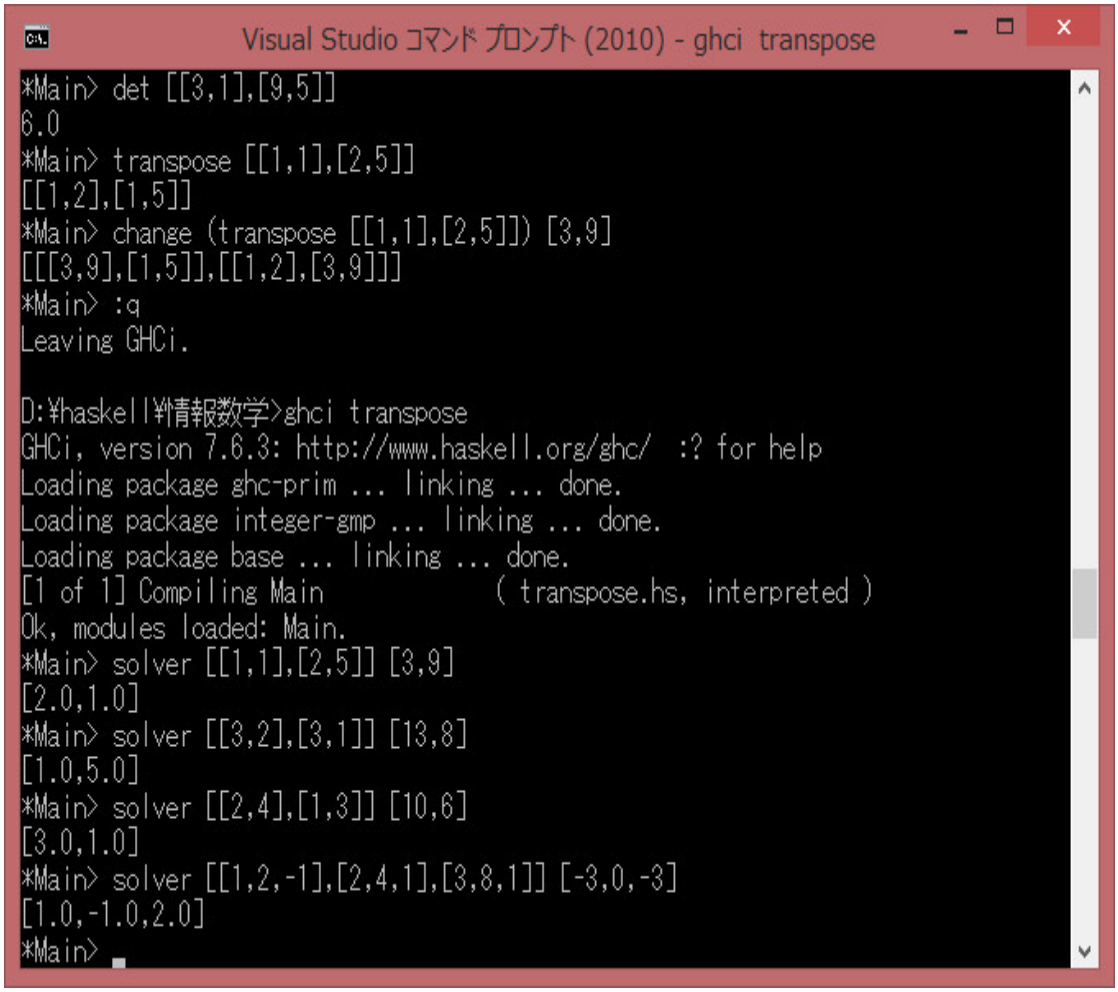
です。

連立方程式を解くプログラムの全体は

```
import Control.Monad

permutation :: Eq a => Int -> [a] -> [[a]]
permutation n xs = iter 0 [] [] where
  iter m ys zs
    | m == n    = reverse ys : zs
    | otherwise = do
      x <- xs
      guard (x `notElem` ys)
      iter (m+1) (x:ys) zs

g :: Int -> [Int] -> Int
g _ [] = 0
g x (y:ys) = if x > y then 1 + g x ys else g x ys

p :: [Int] -> Int
p [] = 0
p (x:[]) = 0
p (x:xs) = (g x xs) + p xs

sign :: [Int] -> Int
sign xs = if even (p  xs) then 1 else -1

h' :: Int -> Int -> [Int] -> [[Double]] -> Double
h' m n js xs
  | m == n    = 1
  | otherwise = ((xs !! m) !! (head js -1)) * h' (m+1) n (tail js) xs

det' :: Int -> [[Int]] -> [[Double]] -> Double
det' _ [] _ = 0
det' n (z:zs) xs = fromIntegral (sign z) * h' 0 n z xs + det' n zs xs
```

```
det xs = det' n perm xs
  where
    n = length xs
    perm = permutation n [1 .. n]

transpose :: [[a]] -> [[a]]
transpose ([]:_) = []
transpose xs = map head xs : transpose (map tail xs)

change' :: Int -> [a] -> a -> [a]
change' 0 (x:xs) y = y : xs
change' k (x:xs) y = x : change' (k-1) xs y
change :: [a] -> a -> [[a]]
change xs y = iter 0 n xs y where
  n = length xs
  iter k n xs y
    | k == n = []
    | otherwise = change' k xs y :iter (k+1) n xs y

solver :: [[Double]] -> [Double] -> [Double]
solver xs ys =
  let zs = transpose xs
      ws = change zs ys
      us = map transpose ws
      d  = det xs
  in map (\a -> (det a) / d) us
```

です。実行して確かめてみましょう。



正しいみたいです。

だいぶ関数型プログラミング言語 Haskell によるプログラミングに慣れてきました。行列式 を行列 の成分とその余因子の積の和にすることを余因子展開すると言います。ついでに余因子展開を使って、帰納的に行列式を計算してみましょう。余因子展開が何かわからない人は佐藤淳郎先生に質問してください。

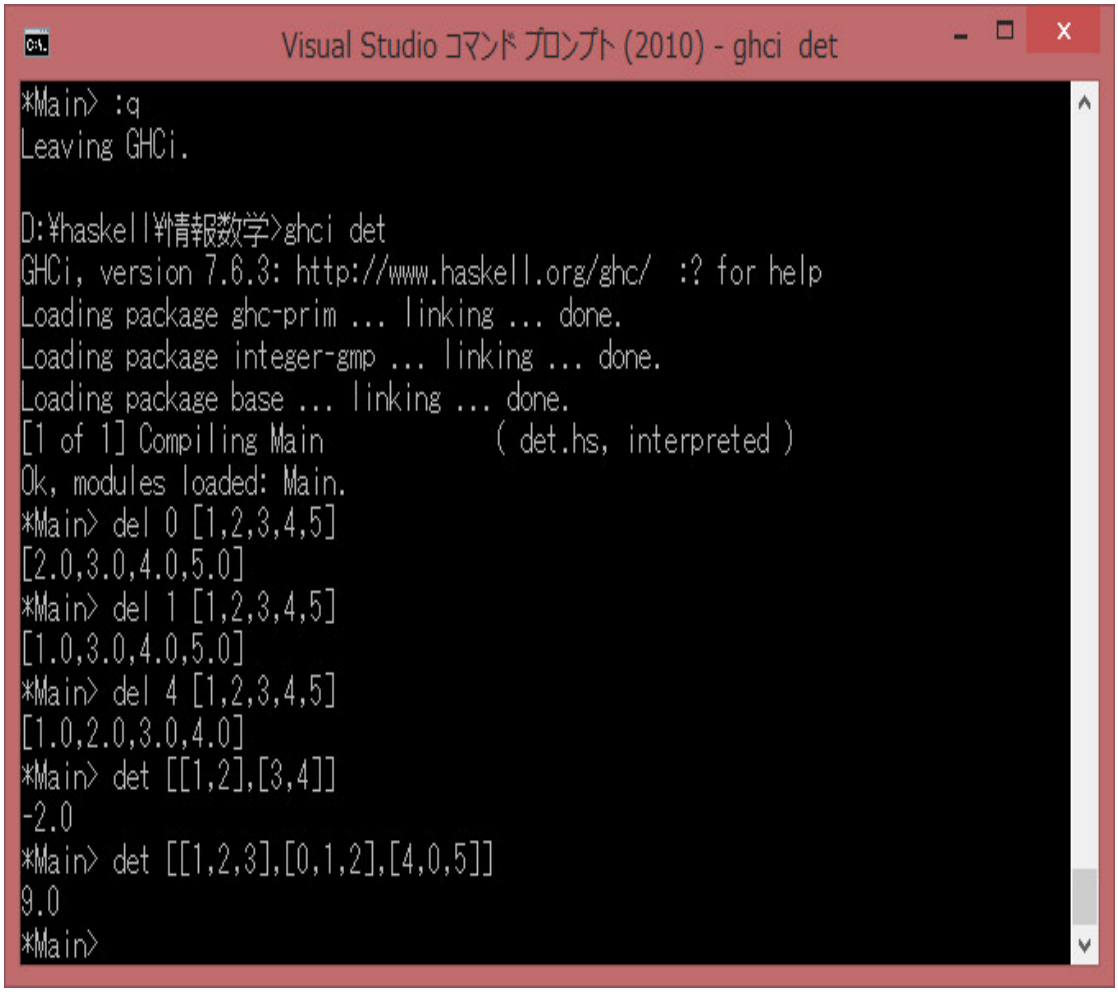
```
del :: Int -> [Double] -> [Double]
del k xs = iter 0 k xs where
  iter i k xs
    | i == k = tail xs
    | otherwise = head xs : iter (i+1) k (tail xs)
det :: [[Double]] -> Double
det [[x]] = x
det xs = iter 0 xs where
  n = length xs
  iter k xs
    | k == n = 0.0
    | otherwise =
```

```
    let x = (head xs) !! k
        ys = map (del k) (tail xs)
        d = det ys
        c = x * d
     in c - iter (k+1) xs
```

がプログラムです。補助関数は del k xs は xs の (k+1) 盤目の要素を削除する関数

$$del\ k\ [a_0, a_1, \cdots, a_k, , \cdots, a_{n-1}] = [a_0, a_1, \cdots, a_{k-1}, a_{k+1}, \cdots, a_n] \quad (0 \leq k \leq n-1)$$

です。正しく動くことを確かめてみましょう。



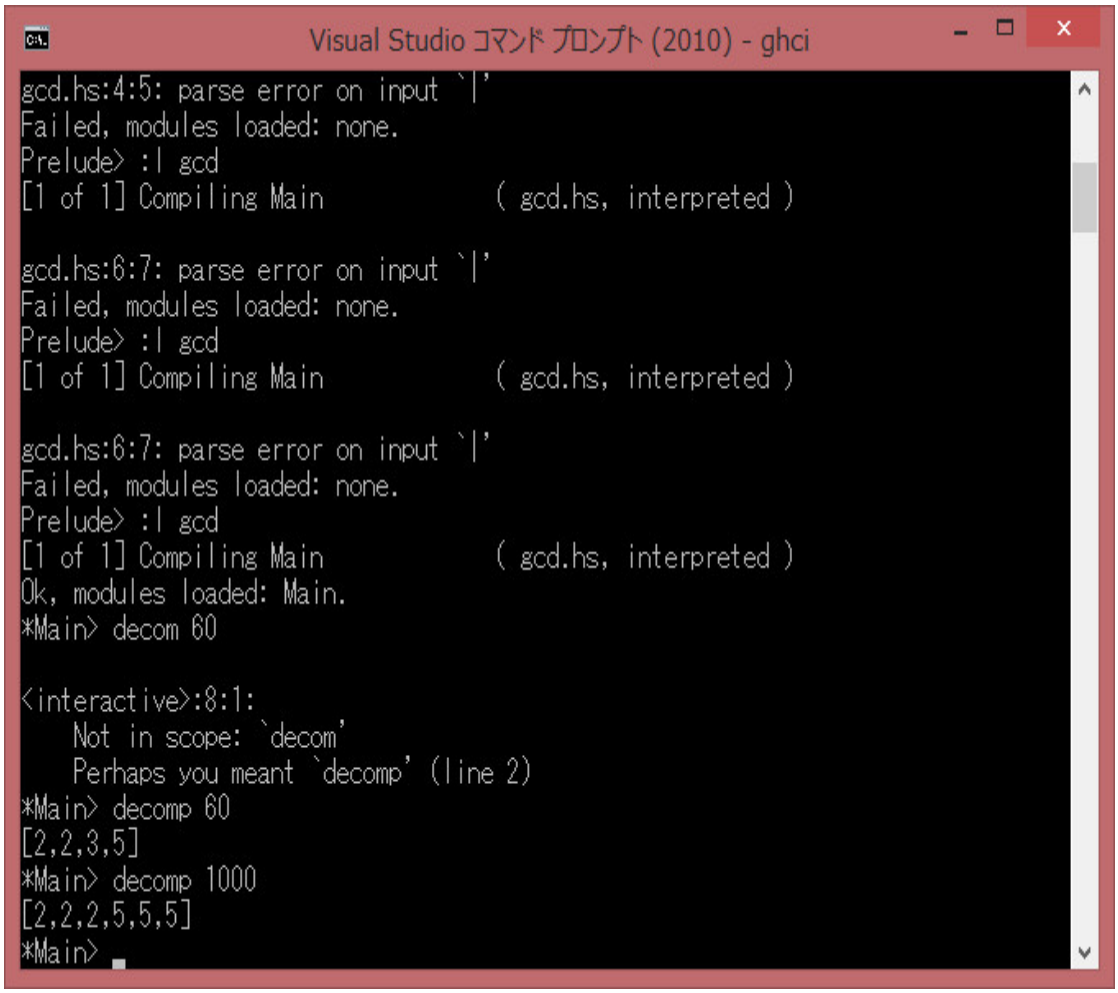
正しいみたいです。行列式の定義を使うより簡単でした。これらのプログラムを眺めていると何か手続き型プログラミング言語の方法論を無理やり関数型プログラミング言語の Haskell に適用しているような気もしてきますが、ともかく Haskell で動くプログラムができました。

最大公約数や最小公倍数は普通ユークリッドの互除法を使いますが、ここでは小学校で習うように素因数分解を使って計算してみます。

まず素因数分解をするプログラムを作ります。例えば、

```
decomp :: Int -> [Int]
decomp n = decomp' n 2 where
  decomp' n m
    | n == 1 = []
    | otherwise = if mod n m == 0 then m : decomp' (div n m) m
                  else decomp' n (m+1)
```

で、計算できます。実行すると

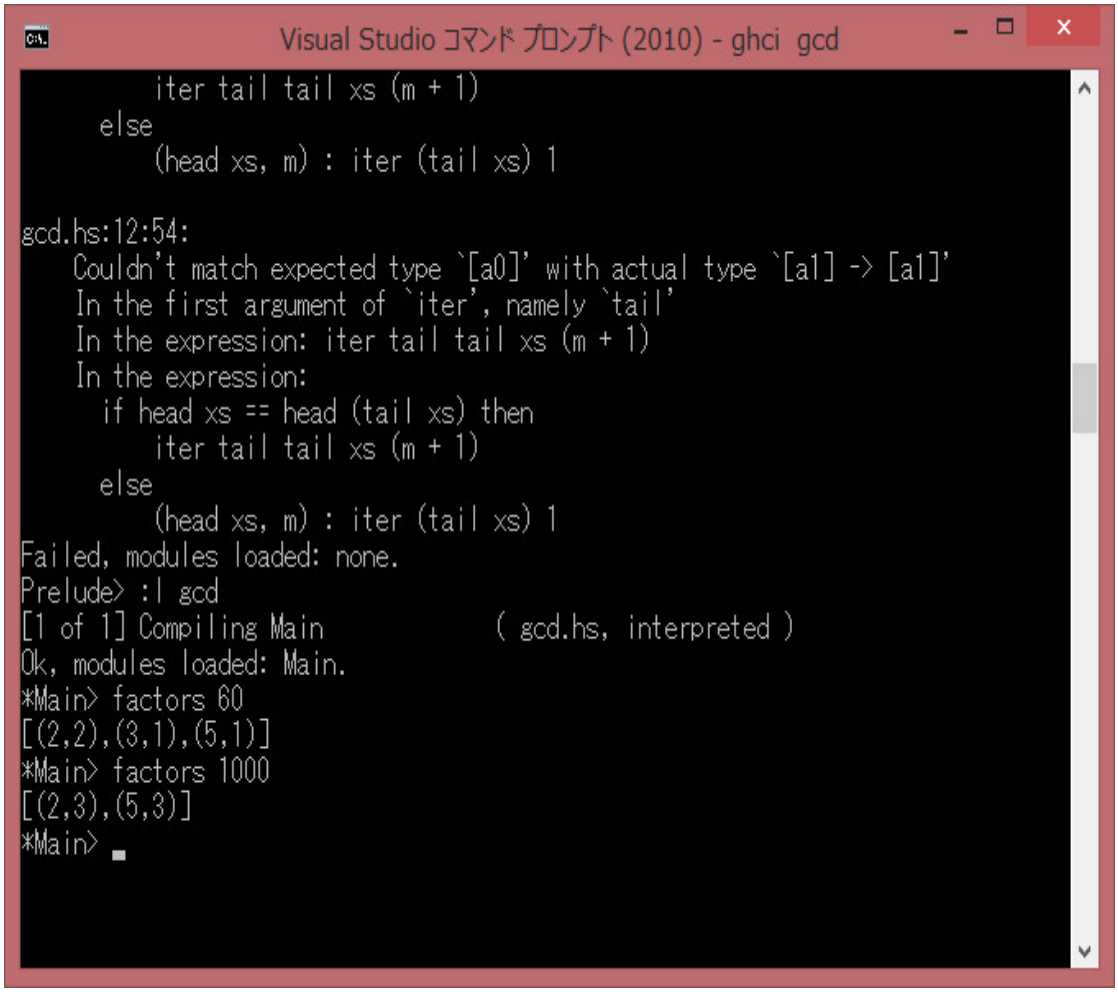


です。*decomp* 60 は $[2,2,3,5]$、*decomp* 1000 は $[2,2,2,5,5,5]$ です。こんな簡単なプログラムでもいっぱい間違っています。間違いを直すことで知識が確実になります。次にこれを $[(2,2),(3,1),(5,1)]$ や $[(2,3),(5,3)]$ の形に変形します。例えば、

```
decomp :: Int -> [Int]
decomp n = decomp' n 2 where
  decomp' n m
    | n == 1 = []
    | otherwise = if mod n m == 0 then m : decomp' (div n m) m
```

```
                  else decomp' n (m+1)
factors :: Int -> [(Int,Int)]
factors n = iter ds 1 where
  ds = decomp n
  iter [] _ = []
  iter [x] m = [(x, m)]
  iter xs m = if head xs == head (tail xs) then iter (tail xs) (m+1)
                  else (head xs,m):iter (tail xs) 1
```

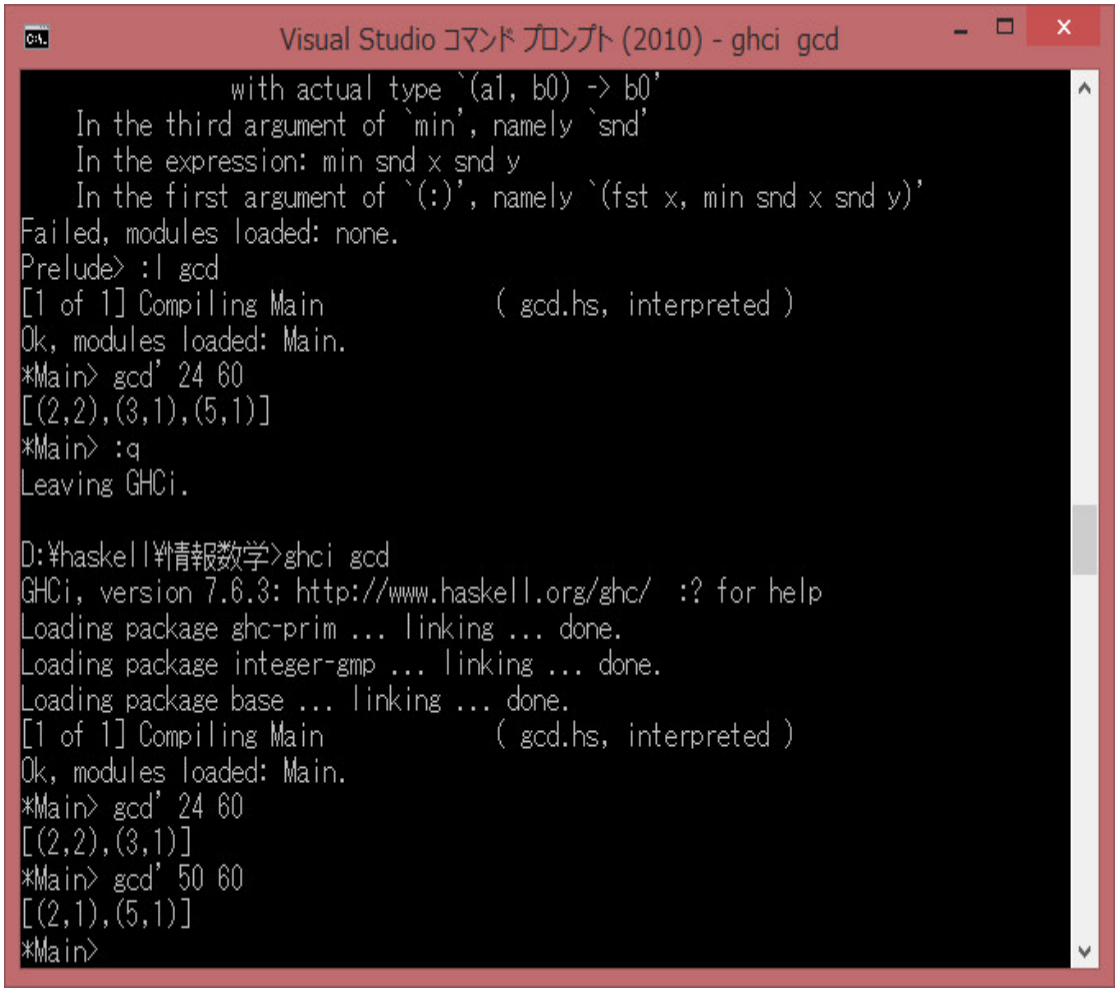
で、計算できます。実行すると



です。$factors\ 60$ は $[(2,2),(3,1),(5,1)]$、$factors\ 1000$ は $[(2,3),(5,3)]$ です。Haskell ではいろいろなプログラミング方法があるのでどの組み合わせを使えばいいか、こんな簡単なプログラムでも試行錯誤を繰り返しています。間違いをちゃんと指摘してくれるので正しくなるまで繰り返します。プログラミングの勉強は、数学や外国語の勉強と同じで基本慣れです。才能のある人にはどうしてもかないませんが、誰でも繰り返しプログラミングしていれば、簡潔で人から絶賛されるようなプログラムは作れなくとも、ともかく正しく動くプログラムをでっちあげることができるようになります。次に、factors を使って共通因数を取り出します。例えば、

```
decomp :: Int -> [Int]
decomp n = decomp' n 2 where
  decomp' n m
    | n == 1 = []
    | otherwise = if mod n m == 0 then m : decomp' (div n m) m
                  else decomp' n (m+1)
factors :: Int -> [(Int,Int)]
factors n = iter ds 1 where
  ds = decomp n
  iter [] _ = []
  iter [x] m = [(x, m)]
  iter xs m = if head xs == head (tail xs) then iter (tail xs) (m+1)
                else (head xs,m):iter (tail xs) 1
gcd' n m = iter ns ms where
  ns = factors n
  ms = factors m
  iter [] _ = []
  iter _ [] = []
  iter uz@(x:xs) zs@(y:ys)
    | fst x == fst y = (fst x, min (snd x) (snd y)) : iter xs ys
    | fst x < fst y  = iter xs zs
    | otherwise      = iter uz ys
```

で、計算できます。実行すると

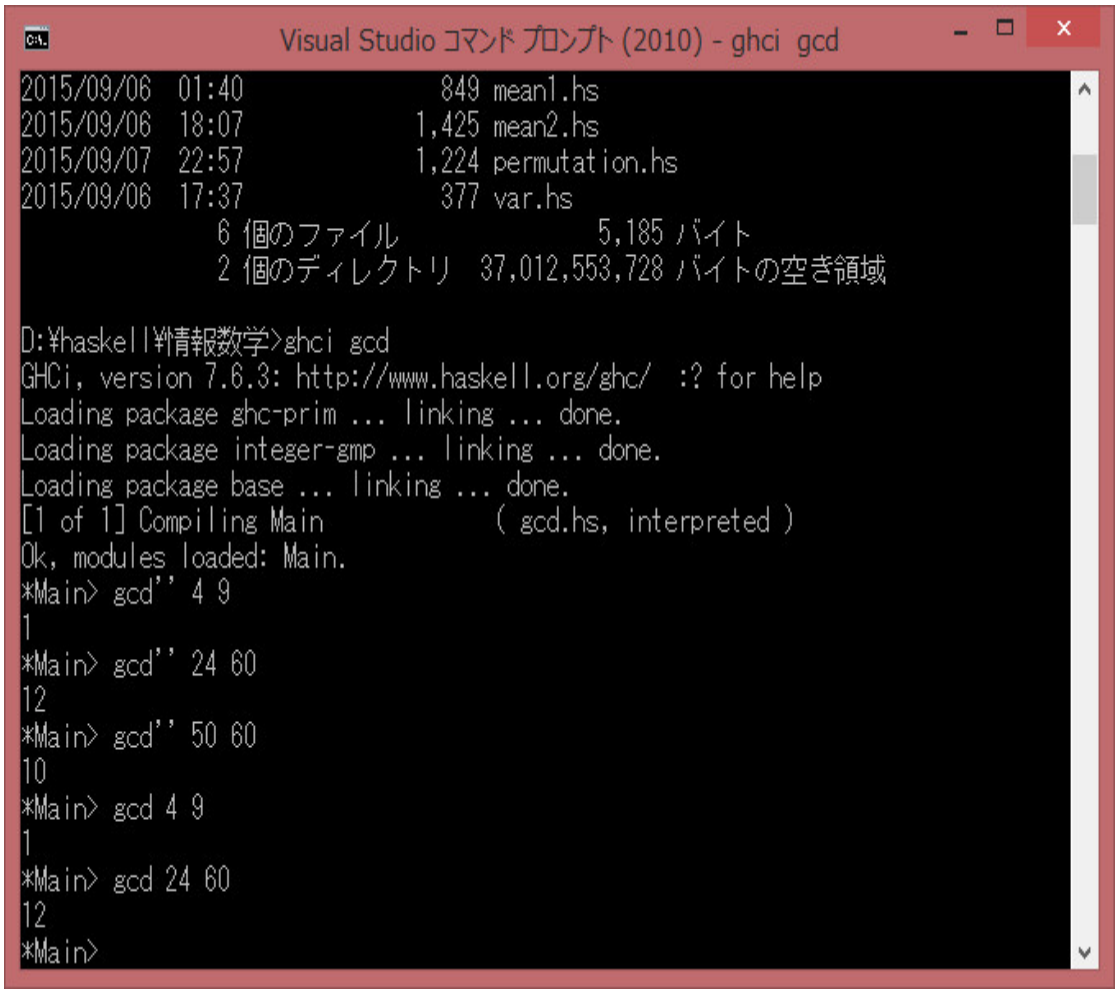


です。 最後に、タプルのリストを整数に変換します。

```
decomp :: Int -> [Int]
decomp n = decomp' n 2 where
  decomp' n m
    | n == 1 = []
    | otherwise = if mod n m == 0 then m : decomp' (div n m) m
                  else decomp' n (m+1)
factors :: Int -> [(Int,Int)]
factors n = iter ds 1 where
  ds = decomp n
  iter [] _ = []
  iter [x] m = [(x, m)]
  iter xs m = if head xs == head (tail xs) then iter (tail xs) (m+1)
                else (head xs,m):iter (tail xs) 1
gcd' n m = iter ns ms where
  ns = factors n
  ms = factors m
```

```
  iter [] _ = []
  iter _ [] = []
  iter uz@(x:xs) zs@(y:ys)
    | fst x == fst y = (fst x, min (snd x) (snd y)) : iter xs ys
    | fst x < fst y  = iter xs zs
    | otherwise      = iter uz ys
gcd'' n m =
  foldl (\a x -> a * ((fst x) ^ (snd x))) 1 $ gcd' n m
```

で、良いです。gcd と言う関数が Haskell にすでに準備されているので gcd" と言う名前にしました。実行すると

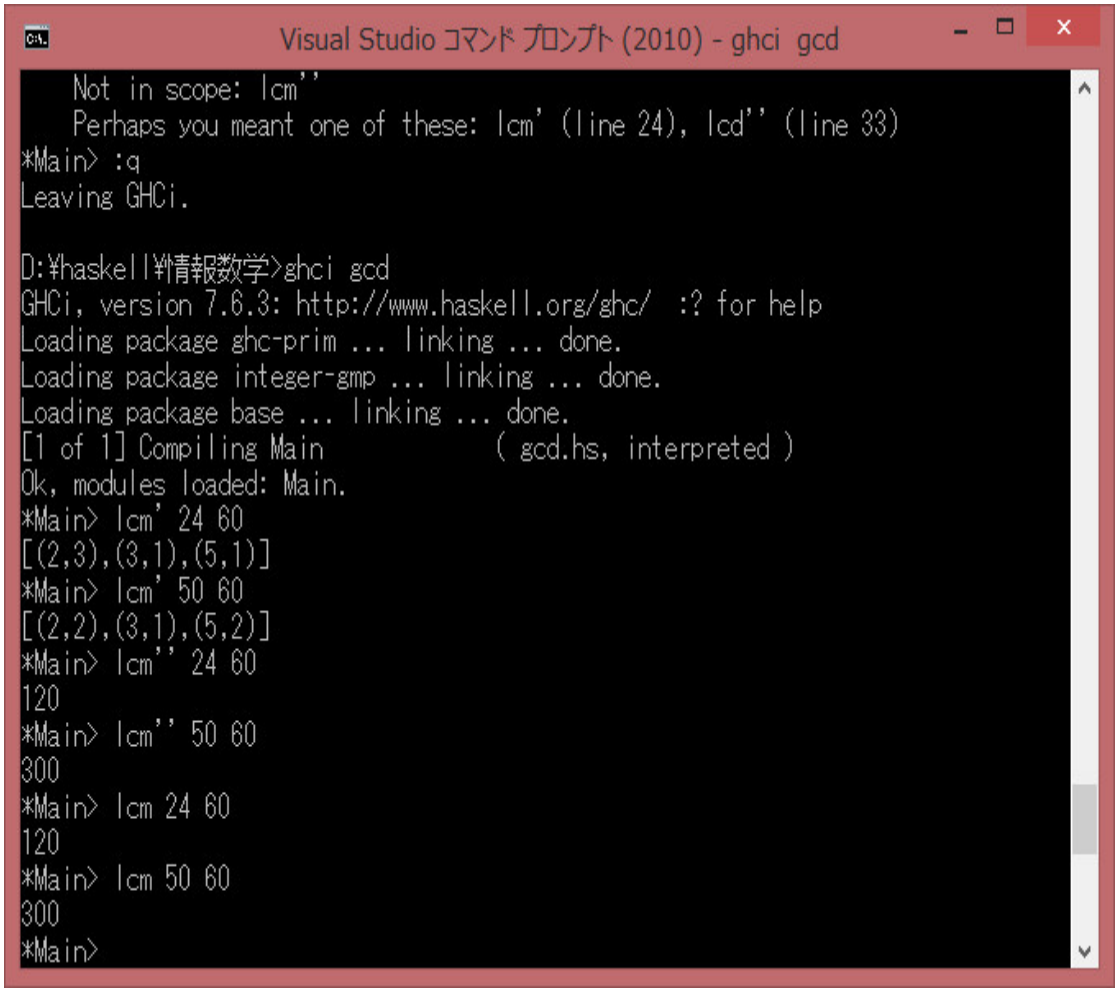


です。次に、最小公倍数を計算します。

```
decomp :: Int -> [Int]
decomp n = decomp' n 2 where
  decomp' n m
    | n == 1 = []
```

```
    | otherwise = if mod n m == 0 then m : decomp' (div n m) m
                  else decomp' n (m+1)
factors :: Int -> [(Int,Int)]
factors n = iter ds 1 where
  ds = decomp n
  iter [] _ = []
  iter [x] m = [(x, m)]
  iter xs m = if head xs == head (tail xs) then iter (tail xs) (m+1)
                 else (head xs,m):iter (tail xs) 1
gcd' n m = iter ns ms where
  ns = factors n
  ms = factors m
  iter [] _ = []
  iter _ [] = []
  iter uz@(x:xs) zs@(y:ys)
    | fst x == fst y = (fst x, min (snd x) (snd y)) : iter xs ys
    | fst x < fst y  = iter xs zs
    | otherwise      = iter uz ys
gcd'' n m =
  foldl (\a x -> a * ((fst x) ^ (snd x))) 1 $ gcd' n m
lcm' n m = iter ns ms where
  ns = factors n
  ms = factors m
  iter [] ms = ms
  iter ns [] = ns
  iter uz@(x:xs) zs@(y:ys)
    | fst x == fst y = (fst x, max (snd x) (snd y)) : iter xs ys
    | fst x < fst y  = x : iter xs zs
    | otherwise      = y : iter uz ys
lcm'' n m =
  foldl (\a x -> a * ((fst x) ^ (snd x))) 1 $ lcm' n m
```

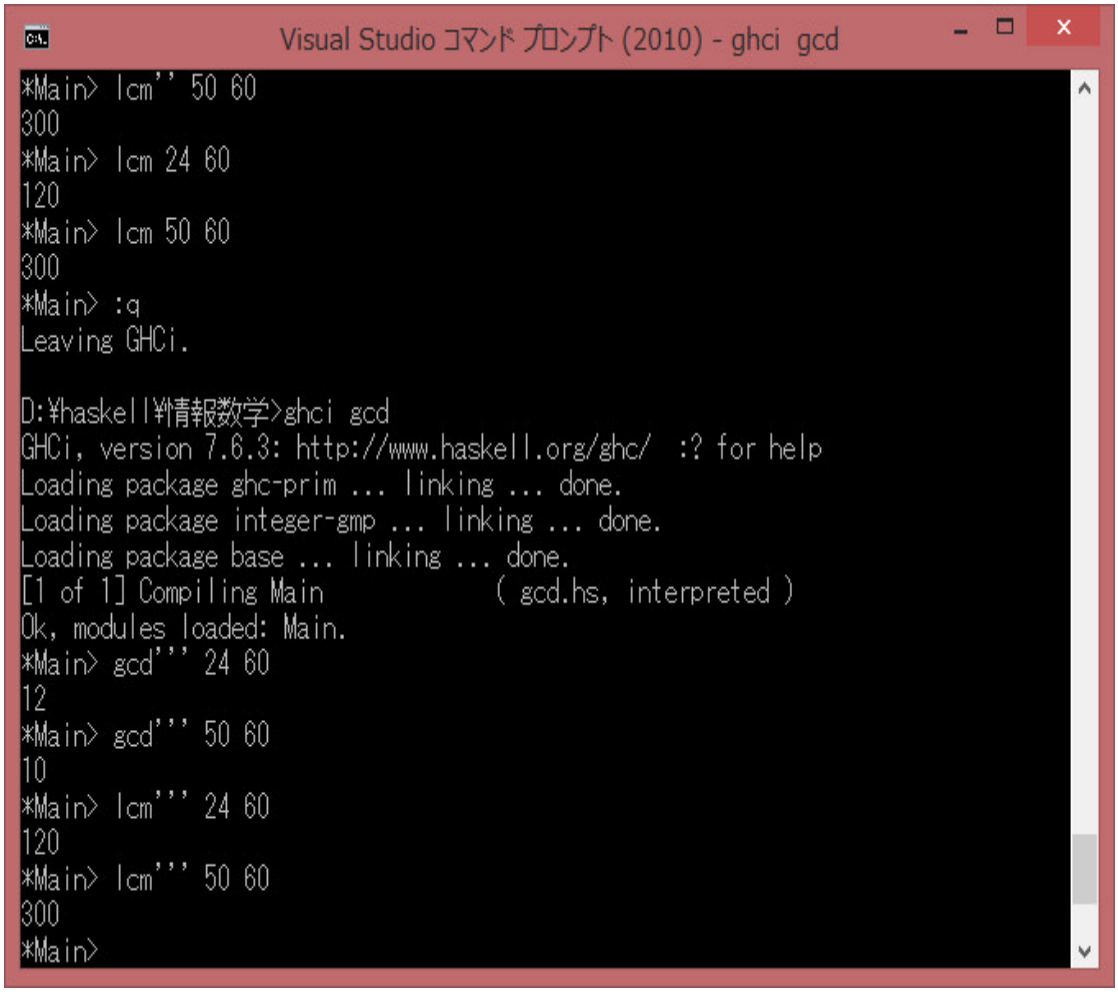
で、良いです。lcm と言う関数が Haskell にすでに準備されているので lcm” と言う名前にしました。実行すると



です。最大公約数や最小公倍数をユークリッドの互除法を使って計算するプログラムは

```
gcd''' n m =
  if m == 0 then n else gcd''' m (mod n m)
lcm''' n m = n * m `div` gcd''' n m
```

です。アルゴリズムの選択の重要性が分かります。実行すると



です。

普段 C++ でプログラミングしているときには、上で述べたようなプログラム（四声の配置や行列式の定義通りのプログラミングや最大公約数や最小公倍数の素因数分解を使った計算など）を作ってみようという気にもなりませんでした。Haskell は面白い言語です。ただ、基本を勉強するだけでも Python の倍以上の時間がかかります。

Haskell の参考書には、「ふつうの Haskell プログラミング」青木峰郎著ソフトバンク、「すごい Haskell たのしく学ぼう！」 Miran Lipovaca 著 Ohmsha、「プログラミング Haskell」 Graham Hutton 著　Ohmsha　などがあります。インターネットにもパズルを解くためのプログラミングを開設したホームページなど色々な情報があります。